Synthetic x-ray imager for solar plasma loops
by Steven Kenneth Lundberg

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
As part of developing a better understanding of the Sun, models describing the magnetic phenomena responsible for the appearance of loops of plasma on the surface of the Sun are being developed. The output from these models is in the form of tables of numbers. Creating images from these tables that look similar to images taken of the Sun in x-ray light would make using and improving the models much easier.

A graphical computer program has been developed to convert the numeric tables from the models into images. One requirement for the program is that it be able to generate images rapidly. The models predict the time-development of the plasma loops, and the images need to be generated and played as a movie. Another requirement of the program is that it run on as many platforms as possible so that it may be used by many researchers.

The images generated are close approximations to the actual plasma loops described by the models. They also look much like actual x-ray images taken of the solar plasma loops. The program is able to generate an image of several plasma loops in less than one minute. This allows the generation of an entire movie consisting of perhaps a hundred images in a few hours. Because C++ and OpenGL are available on both Windows and Unix platforms, the program can be compiled and run on many of the machines used by researchers around the world.

The program works as intended and will be a substantial help in the further development of the solar magnetic models. It is general enough that it can be used by many different researchers using different solar magnetic models.

SYNTHETIC X-RAY IMAGER FOR SOLAR PLASMA LOOPS

by

Steven Kenneth Lundberg

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

May, 1998

APPROVAL

of a thesis submitted by

Steven Kenneth Lundberg

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

4/9/98 — Date

J Dunbigh Stanley — Chairperson, Graduate Committee

Approved for the Major Department

4/9/98 — Data

J Dunbigh Stanley — Head, Major Department

Approved for the College of Graduate Studies

4/15/98 — Date

Joseph J. Fedock — Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Steven K. Lundberg

Date April 9, 1998

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

As part of developing a better understanding of the Sun, models describing the magnetic phenomena responsible for the appearance of loops of plasma on the surface of the Sun are being developed. The output from these models is in the form of tables of numbers. Creating images from these tables that look similar to images taken of the Sun in x-ray light would make using and improving the models much easier.

A graphical computer program has been developed to convert the numeric tables from the models into images. One requirement for the program is that it be able to generate images rapidly. The models predict the time-development of the plasma loops, and the images need to be generated and played as a movie. Another requirement of the program is that it run on as many platforms as possible so that it may be used by many researchers.

The images generated are close approximations to the actual plasma loops described by the models. They also look much like actual x-ray images taken of the solar plasma loops. The program is able to generate an image of several plasma loops in less than one minute. This allows the generation of an entire movie consisting of perhaps a hundred images in a few hours. Because C++ and OpenGL are available on both Windows and Unix platforms, the program can be compiled and run on many of the machines used by researchers around the world.

The program works as intended and will be a substantial help in the further development of the solar magnetic models. It is general enough that it can be used by many different researchers using different solar magnetic models.

INTRODUCTION

Problem Background

Physicists are studying the Sun in an attempt to understand the processes at work there (Longcope). One of the tools they use is x-ray pictures taken of the corona. The plasma in the corona is hot enough that it emits x-rays, so features in the corona such as the plasma loops there can be seen in x-ray photographs. The plasma loops in the corona are driven by convection and other transport processes occurring within the Sun. Some understanding of the processes at work below the surface of the Sun can be gained by studying the loops visible in an x-ray picture. Since x-rays do not penetrate the Earth's atmosphere, the x-ray pictures of the Sun must be taken from satellites.

The work described here was done in conjunction with the MSU Solar Physics group led by Dr. Loren Acton. Dr. Acton's group is one of several research organizations using the Yohkoh satellite to take x-ray pictures of the Sun. The Yohkoh satellite is the product of a joint Japanese - U.S. venture. This satellite is operated by Lockheed and current (generally less than 24 hours old) x-ray pictures of the Sun are posted on the Internet (Lockheed). Dr. Acton, then at Lockheed, directed the design and construction of the Soft X-ray Telescope (SXT) carried on YOHKOH.

Solar Magnetic Phenomena

Much of what can be seen in x-ray pictures of the Sun is believed to be the result of magnetic fields interacting with the plasma in the Sun's corona. These magnetic fields are believed to be driven by electrical currents within the Sun. The electrical currents are the result of convective processes occurring inside the Sun. As the magnetic fields interact with the coronal matter, the matter is transported and heated. The pattern of heating determines the quantity and distribution of x-ray emission.

Magnetic field enters the corona from the interior of the Sun through isolated magnetic features on the solar surface. These features are the top part of magnetic features (called flux tubes) from the interior of the Sun. Often these features are connected by magnetic field lines within the corona. They influence each other through these connections. One possible configuration is with two magnetic features opposing each other (e.g. North to North). The features can be driven toward each other by the underlying dynamics within the Sun, but they repel each other through their magnetic field interaction. Energy can be stored in this interaction. When the energy stored reaches some threshold, the two magnetic fields reconnect their field lines in a new combined configuration. This releases much of the energy stored in the previous configuration. This energy gets deposited as heat in the plasma where the field lines are. The increased heat in the plasma makes it emit more x-rays. These x-ray emissions are the probe being used to follow the magnetic field interactions. Plasma is in the magnetic field lines above the Sun's surface because as the magnetic field lines are pushed up

from within the Sun, they drag the plasma along with them. This occurs because charged particles cannot readily cross magnetic field lines.

Understanding Solar Processes

One common way to formulate our understanding of physical processes is to build models of those processes. When these models closely approximate what happens in nature, we believe the models to be accurate. In the case of understanding solar dynamics, models of the formation and time development of plasma loops and systems of interacting plasma loops have been developed. However the models are numerical models and it is difficult to compare the tables of model output with x-ray images taken of plasma loops on the Sun. One very powerful method for comparing the models to actual events is to create synthetic x-ray pictures of the loops in the models and compare these pictures to the ones taken of the Sun. However, at present only relatively crude methods are used to generate pictures of these model plasma loops (Reale). The value of good color renditions of the plasma loops generated by the models is readily apparent. The opportunity to develop a program that would be of great value to Dr. Acton's group, and others in the solar physics community, resulted in the Synthetic X-ray Imager (SXI) computer program described in this thesis.

SXI REQUIREMENTS

The primary goal of the SXI program is to support the development of solar magnetic models. As such, it needs to have an easy and natural interface. SXI is designed to use the same parameters and measurement units as the solar magnetic models. Since SXI is intended to create images that look similar to the ones taken by the x-ray camera on the Yohkoh satellite, it also uses parameters and units typical of those used on that satellite.

The solar magnetic models describe the plasma loops they are dealing with by a series of points and associated attributes. Each point in space has an associated radius, a density factor, and a temperature factor. By connecting the points with an envelope that extends a radius from each point, a three-dimensional loop is generated. Since the plasma density and temperature typically vary along the length of the plasma loop, these properties are also associated with each point. SXI averages the density and temperature attributes at each two connected points to obtain the values used for that portion of the loop.

The camera in the satellite uses different exposure times and filters to effectively image different parts of the solar plasma loops. SXI uses the measured properties of the filters in Yohkoh to create images that embody the same responses to x-ray energy.

Yohkoh images the Sun with a CCD camera. This yields pictures with a fixed

pixel resolution. SXI is designed so that the pixel size in the images can be set to any desired size, or allowed to adjust so that the entire plasma loop is within the image.

One primary aspect of the images taken by Yohkoh and modeled by the solar magnetic models is the time development of the plasma loops. Through interactions with other loops and under control of the transport phenomena inside the Sun, the plasma loops grow, merge, move, and change temperatures. This dynamic aspect of the plasma loops is particularly important for verifying the model accuracy. Thus SXI is designed to be able to generate images quickly enough to create a movie with one hundred images on a time scale measured in hours, not days or weeks.

Because SXI is intended for use by solar physicists at perhaps several institutions it is written in a combination of the C++ language and OpenGL. Both C++ and OpenGL are available on many platforms. As a result, SXI runs under both Windows95 and Unix.

SXI INTERNALS

The actual code for SXI is included as Appendices A-F. Some of the more interesting aspects of the operation of SXI are discussed below.

Building the Plasma Loop

The loop is constructed from common building blocks. Since the radius of the loop can change along the length of the loop, tapered cylindrical shapes are used. Each cylinder is truncated at both ends to fit neatly against the next cylinder in the loop. This process requires many steps. The function MakeCylinder() takes two DISKs each containing a center position and radius to mark the bottom and top of the cylinder. A DISK is a structure defined in the file sxi.h that is made up of the elements: center (of type POINT3); radius; density; and temperature. A POINT3 is a structure made up of three values for the x, y, and z coordinates of the point. MakeCylinder() also takes two more DISKs containing the prior and subsequent points and radii. (These extra DISKs are used to determine the planes which will cut the ends of the cylinder so that it will fit neatly to the next one in line.) It makes a tapered cylinder using the DISKs for the bottom and top of the cylinder. Then the cylinder is lengthened and trimmed using a cut plane calculated from the two points that surround the point used for each end (see Figure 1). This cut plane is defined as the plane perpendicular to the line connecting the

two surrounding points and containing the end point.

The OpenGL procedure gluCylinder() is used to build each cylinder. (In OpenGL cylinders can have different radii at each end.) However, since gluCylinder() creates cylinders with one end at the origin and the axis along the positive z-axis, each piece of the loop must first be translated so that its initial end is at the origin and then rotated so that its axis lies along the z-axis. Then the cylinder can be created. The cylinder is lengthened as mentioned above to make sure that the cut planes lie entirely within the cylinder.

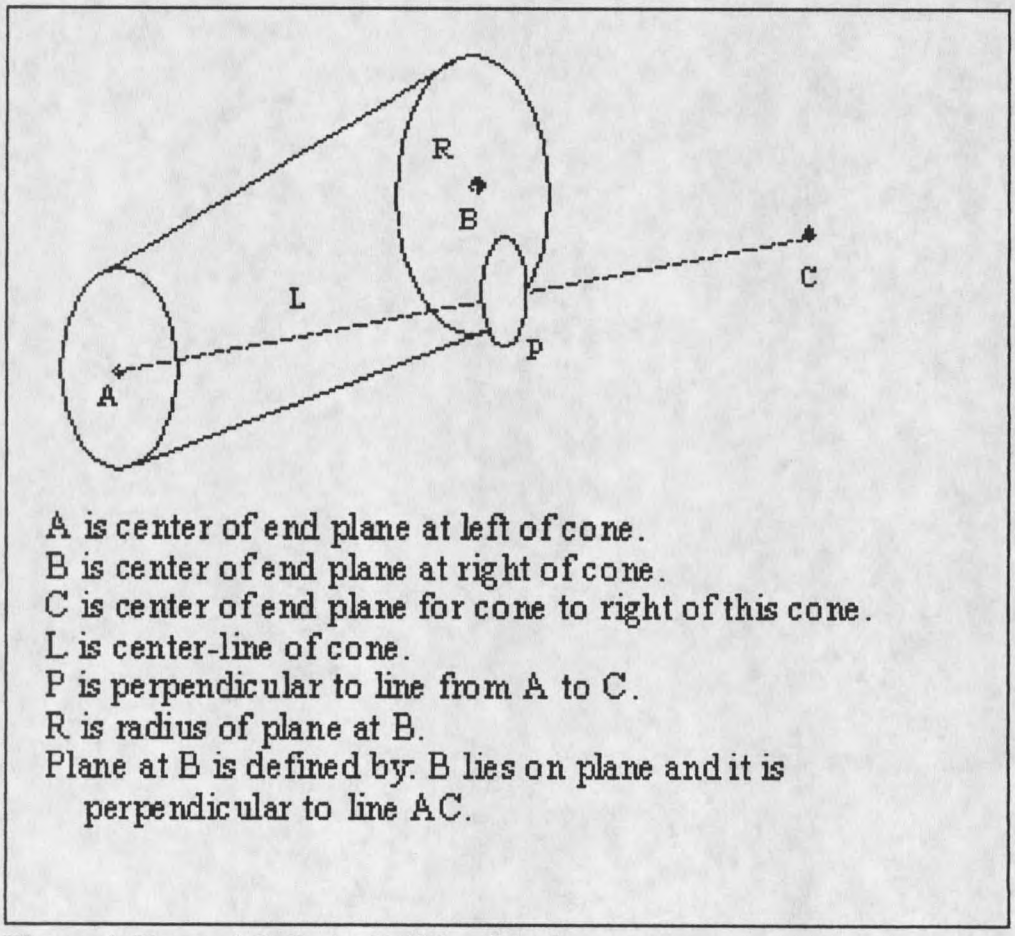

Figure 1: Tapered Cylinder Building Block

After the cylinder is created, it is placed back in the image where it originally was, by reversing the rotation and translation operations. This reversal is planned for by saving the projection matrix before moving the cylinder to the origin. After creating the cylinder, simply restoring the previous projection matrix puts the cylinder back where it belongs. For each set of four records in the input data file, one cylinder can be drawn. The process proceeds by sliding over one record and producing another cylinder until the end of the data file is reached. This means that each record in the data file, except the ones at the beginning and end, is used as an end twice (for each of the cylinders that meet at that point) and four times to help define a cut plane.

Mapping to Pixel Colors

After each tapered cylinder is drawn it is mapped to pixel colors. The color of each pixel is proportional to the depth, density, and temperature of the portion of the plasma loop that maps to that pixel. Here depth means the distance through the plasma loop as seen from the pixel in question. The process requires several steps. To begin with, this is an area where designing for speed is critical. We are about to map this cylinder on a pixel by pixel basis and do not want to deal with pixels that do not map from this cylinder.

The depth is the distance through the cylinder as experienced by a ray emanating from the pixel on the screen and traveling perpendicular to the screen. The density value

used is actually the square of the electron number density divided by 10^{20} which is read directly from the input data file. The temperature value is a function of the plasma temperature. The temperature variable is x-ray luminosity convolved with the response function of the filter and recorder being simulated. (X-rays from plasma of different temperatures have different characteristic wavelengths and each filter material has a different response to x-ray wavelength.) The temperature variable is calculated by using the temperature in degrees Kelvin read from the input file and a function built in to the program. The function calculates the expected number of counts in an individual detector bin from plasma of the given temperature and normalized for exposure time, electron number density, depth of the plasma path, and the area on the Sun that is imaged by the specific detector bin. When this value is multiplied by the actual exposure time, electron number density, and plasma path depth it gives the expected counts per pixel for the synthetic x-ray image.

One further adjustment needs to be made. The Yohkoh camera uses a charge coupled device (CCD) to collect the x-ray photons and the counts are digitized to 12 bit accuracy. This program is using 8 bit values to store the pixel colors. To make the SXI images saturate at the same x-ray intensity as the Yohkoh camera the counts per pixel are divided by 16 to allow for the greater dynamic range of Yohkoh.

Mapping Transformations

The mapping process requires both coordinate transformations and intersection calculations. The coordinate systems will be discussed first.

The final synthetic x-ray image must be drawn in screen coordinates, but the individual cylinders are initially drawn symmetric about the z-axis and their equations of form are relatively simple in that coordinate system, hereafter called the cylinder coordinate system.. The rays emanating from the screen are known in screen coordinates, but must be intersected with the cylinder in cylinder coordinates. Thus the screen ray must be transformed to cylinder coordinates. Since this transformation involves only translation and rotation, any distances will remain unchanged.

Once the screen ray is in cylinder coordinates, it must then be tested to see if it intersects with the truncated, tapered cylinder. If it does intersect, then the distance through the cylinder is calculated. The OpenGL Programming Guide (Woo) was the primary reference used for the OpenGL-specific functions and for the various coordinate transformations.

The steps proceed as follows:

First, calculate the direction vector from the screen into the scene. Since we are using parallel projection there is only one direction vector for the entire scene.

Second, calculate the bounding rectangle for the cylinder in screen coordinates (Foley 462-465). This is done by placing four points around the ends of the un-clipped cylinder. The positions of the four points are at square root of 2 times the radius of the end of the cylinder along both the x and y axes at z equals 0 and at z equals the cylinder height. The square root of 2 times the radius makes certain that no matter what viewing angle about the z-axis is used, the points lie outside of the image of the cylinder end. These eight points are transformed to screen pixel coordinates and the maximum and

minimum x and y values among all eight points are determined. These maximum and minimum x and y values then determine a bounding box for the image of the cylinder in screen coordinates (see Figure 2).

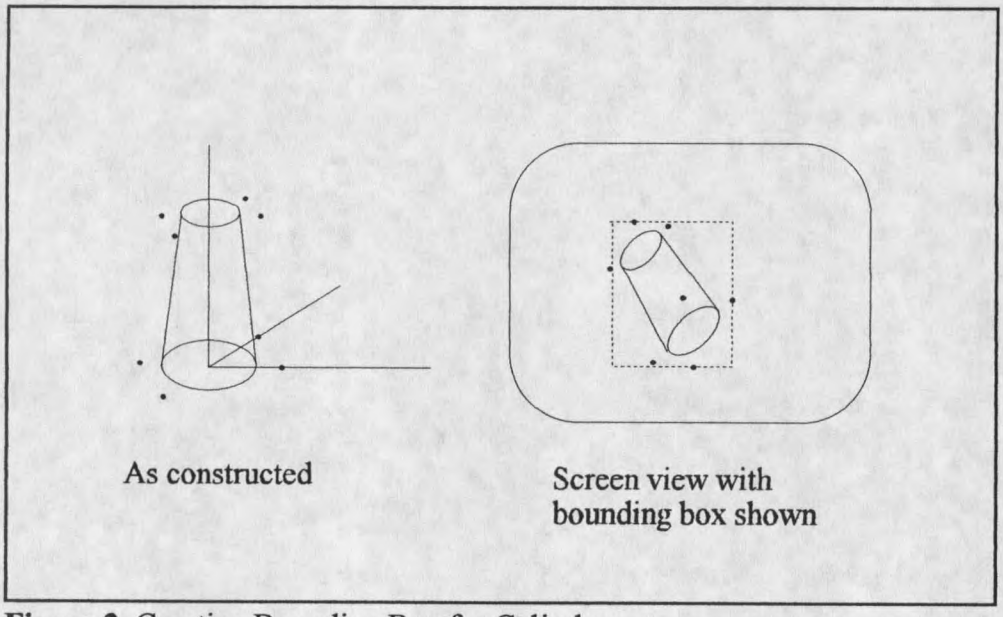

Figure 2: Creating Bounding Box for Cylinder

Third, for each pixel in the bounding rectangle, determine if the ray emanating from it intersects with the cylinder. This requires mapping the screen pixels and rays to the cylinder coordinates, then calculating the intersections.

Fourth, if the ray intersection with the cylinder is outside one of the clip planes for the ends of this cylinder, replace this intersection with the intersection of the ray with the clip plane. But, if the second ray intersection with the cylinder is also outside the same clip plane, then this ray doesn't intersect the clipped cylinder at all (see Figure 3).

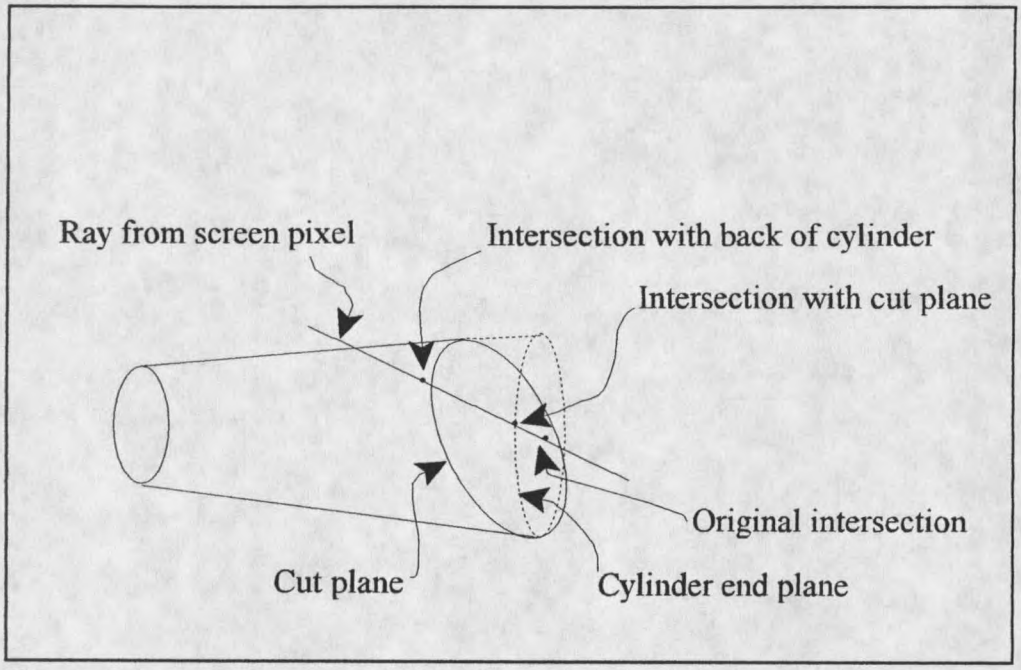

Figure 3: Ray Intersection with Cut Plane

Fifth, calculate the distance between the two points of intersection.

Sixth, set the x-ray image buffer pixel color proportional to the distance.

All of the coordinate transformations are accomplished by matrix multiplication. The transformation from cylinder coordinates to screen coordinates uses just the projection transformation already calculated by OpenGL. The inverse of that matrix is used to go from screen to cylinder coordinates. The inverse matrix is calculated by the Gauss-Jordan Elimination method (Press 32-37).

The surface of a tapered cylinder can be represented by a mathematical construct known as a quadric. This is a 4 x 4 matrix. The values in the matrix are determined from the equation of the surface:

$$x*x + y*y - r1*r1 + 2r1*(r1-r2)(z/zr2) - (r1-r2)^2(z/zr2)^2 = 0,$$

where r1 and r2 are the radii at the two ends of the tapered cylinder. The cylinder has one end at z = 0 and the other end at z = zr2. The general form of the quadric equation is (Kirk 280):

$F(x,y,z) = ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fxy + 2gy + hz^2 + 2iz + j = 0.$

The matrix coefficients are: Q11 = a; Q12 = Q21 = b; Q13 = Q31 = c; Q14 = Q41 = d; Q22 = e; etc. For a right tapered cylinder, with z the axis of symmetry, the values of the non-zero elements in Q are: Q11 = Q22 = 1; $Q33 = -(r1 - r2)^2/h^2$; Q34 = Q43 = $(r1^2 - r1r2)/h$; and $Q44 = -r1^2$. Where r1 is the radius at z = 0, r2 is the radius at z = h, and h is the height of the cylinder. So only Q33, Q34, and Q44 need to be calculated for each cylinder.

To calculate the intersections, we need to solve the simultaneous equation for its roots, which are the intersections. We need the equation for the cylinder in question in cylinder coordinates - this is the quadric. The equation to be solved is in the form of: $k2*t^2 + k1*t + k0 = 0$, where t is the variable that determines how far along the ray we must go to find an intersection. For the case of a right tapered cylinder (called cylinder for short):

k2 = ax*ax*Q11 + ay*ay*Q22 + az*az*Q33;

k1 = 2*ax*x0*Q11 + 2*ay*y0*Q22 + 2*az(z0*Q33 + Q34);

k0 = x0*x0*Q11 + y0*y0*Q22 + z0*z0*Q33 + Q44 + 2*z0*Q34;

The Q's are the non-zero elements of the quadric form of a right tapered cylinder, ax, ay, and az are the direction vector elements for the ray, and x0, y0, and z0 are the starting end-point values for the ray.

Image Storage

A 256 color lookup table is defined for the synthetic x-ray image colors. Each pixel color is stored as one 8-bit byte. The value of the pixel byte is used to index the lookup table. The pixel values are stored in an array with the lower left pixel in array position 0. This pixel array is exported as a bitmap file for use by other programs. An example of a bitmap file (converted to grayscale) is shown in Figure 4.

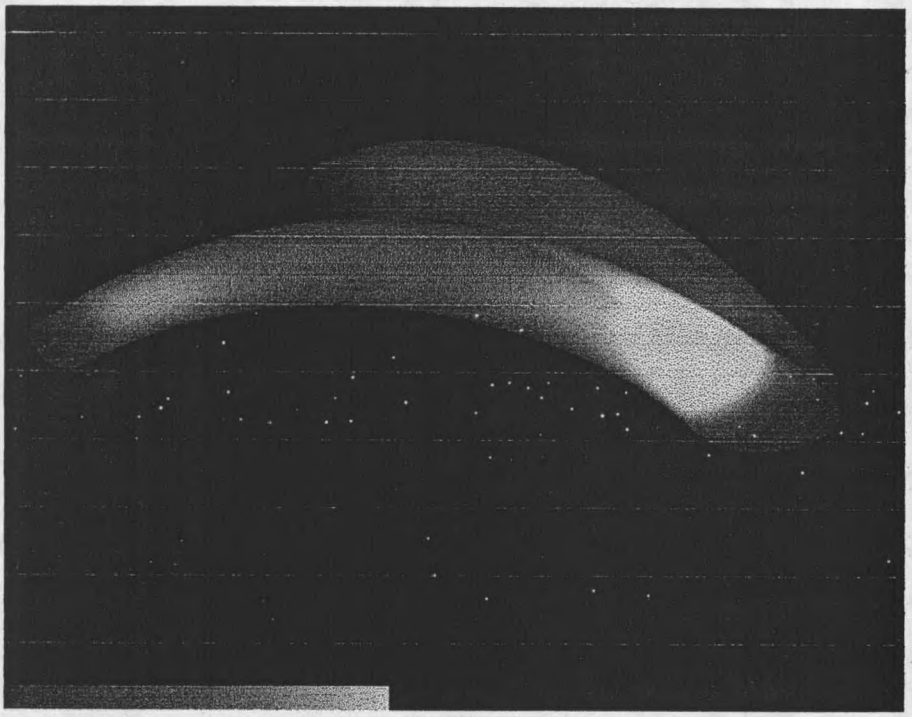

Figure 4: Grayscale Version of Synthetic X-Ray Image

CONCLUSION AND POSSIBLE IMPROVEMENTS

The SXI program meets its initial requirements:

- It draws a synthetic solar plasma loop from the output of a mathematical model of solar dynamics.
- The time it takes is short enough to support making movies of the time development of these plasma loops.
- The input is given in units typically used for solar modeling.
- It supports plasma loops where the temperature and density vary along the axis of the loop.
- The pixel size can be fixed at some desirable size, or allowed to adjust to make the plasma loop fit in the viewing area.
- It synthesizes images using different camera filters and sensitivities.
- It is portable between Windows and Unix environments.

Several modifications may enhance the usability of SXI:

- Currently the viewing volume is centered on the origin. Removing this requirement would allow easier interface to model output.
- Some textual output on the synthetic image would be desirable. This might include data identification, filter in use, and exposure time.
- The coordinates could be changed from Cartesian to spherical to be more consistent with the usual solar coordinate use.

SXI USER'S MANUAL

SXI uses two input files: the data file and the initialization file. The data file must be named on the command line while the initialization file has a fixed name and must reside in the current working directory.

The data file format is one six-entry record per line. The six numeric values in each record are: radius, x-coordinate, y-coordinate, z-coordinate, density value, and temperature. The radius and coordinates must be given in the same units, but the choice of units is arbitrary. The density value is the square of the electron number density divided by 10^{20} which is expected to be a number between zero and 100. The temperature is given in degrees Kelvin. The values must be separated by white space and terminated with an end-of-line. They will be read in floating point format and can be in decimal notation or exponential (base ten) notation. If more than one plasma loop is contained in the file the special string "END LOOP" must be placed on the line immediately following the last record for each loop. The next record will be taken as the first record for a new loop. The first and last records for each loop are only used to determine the cut plane for the ends of the plasma loop. They do not contribute to the loop itself.

The initialization file is named "sxi.ini". It has four lines of setup information. The file format is:

line 1: windowX, windowY, windowWidth, windowHeight

line 2: EYEX, EYEY, EYEZ, CENTERX, CENTERY, CENTERZ, UPX, UPY, UPZ

line 3: Sensitivity function index, exposure value, pixel sizing, size

line 4: xOffset, yOffset, zOffset

Line 1 controls the position and size (in pixels) of the window on the screen. This needs to be the same size as the final synthetic x-ray image or pixel artifact effects will be seen. Currently the synthetic x-ray image is 600 by 600 pixels. Line 2 controls the viewpoint from which the scene is drawn. The three EYE values are the coordinates of the viewing point. The three CENTER values are the coordinates of the center of the scene. The three UP values determine the direction shown as the top of the scene on the screen. The CENTER values are designed to be the origin. The EYE values work best if they are each less than 1. The CENTER and EYE values really only determine the direction from which the scene is viewed. (See the explanation for line 4 below.) Line 3 controls the x-ray picture generation. The sensitivity function index sets the sensitivity function to be used. This relates to the type of filter in use. The exposure value sets the exposure which relates to the brightness of the image. Pixel sizing defines the type of scaling to be used. FIXED_AREA_SIZE will show only that part of the image that is within the fixed boundaries. FIXED_PIXEL_SIZE will show the image as it would look with each pixel covering the stated amount of area on the Sun's surface. If line 3 has FIXED_AREA_SIZE, then size contains the horizontal span for the fixed area - the y and z values match the x one to make the viewing space a cube. If line 3 has FIXED_PIXEL_SIZE then the value of size is the span of one pixel. Line 4 is the offset

for the center of the viewing volume. The x, y, and z values entered here will be the center of the volume. NOTE: All length parameters - radius, x, y, z, pixel size, and offset must be in the same units. Positive z is away from the Sun's surface.

Examples of sxi.ini and a data file are included in Appendix G. An example of the program command line would be "sxi input_data" where input_data is the name of the file containing the loop data and the file sxi.ini is in the curent working directory.

REFERENCES CITED

Foley, VanDam, Feiner, Hughes, and Phillips, *Introduction to Computer Graphics*, Reading, Massachusetts, Addison-Wesley Publishing Company, 1995.

Kirk, D. (ed.), *Graphics Gems III*, Boston, Harcourt Brace Jovanovich, c1992.

Lockheed, First_Light, [online] Available http://www.space.lockheed.com/SXT/html2/First_Light.html. March 24, 1998.

Longcope, D. W., "Topology and Current Ribbons: A Model for Current, Reconnection and Flaring in a Complex, Evolving Corona", *Solar Physics*, 169(1996), 91-121.

Press, W. H., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge, Cambridge University Press, 1988.

Reale, F. and Peres, G., "Solar Flare X-ray Imaging: Coronal Loop Hydrodynamics and Diagnostics of the Rising Phase", *Astronomy and Astrophysics*, 299(1995), 225-237.

Woo, Neider, and Davis, *OpenGL Programming Guide Second Edition*, Reading, Massachusetts, Addison-Wesley Developers Press, c1997.

APPENDICES

APPENDIX A

SXI.CPP

```
/*=============================================================================

sxi.cpp    Steve Lundberg    2/15/97

This is the driver file for viewing the intensity of x-rays produced by
solar flares defined by solar magnetic models.

It produces synthetic xray pictures from the size, density, and temperature
of the loops.

This program is capable of rendering several plasma loops in one image.

Although this code is written to be compiled with a C++ compiler, nearly
all of the code is standard C.

=============================================================================*/
/*****************************************************************************

Process:
A
 Use the points defined in the input data, together with the radii also
 defined in the input data, to produce tapered cylinders with ends cut at
 an angle that neatly meets the next cylinder.
B
 From the view-point of the screen view, for each screen pixel that is in
 a cylinder:  Color that pixel proportional to the distance through the
 cylinder in the direction from the screen pixel perpendicular to the
 screen.  Another way to say this is that the pixel is colored proportional
 to the distance a ray from the screen pixel and perpendicular to the
 screen will travel from the point where it enters the cylinder to the
 point where it leaves the cylinder.  Since all cylinders are convex
 objects, no ray can go through more than one place on an individual
 cylinder.  It is possible, however, for a given ray to penetrate more
 than one cylinder.  This simply means that the pixel color will be
 determined by the total depth of all cylinders that ray passes through.

 1
  These rays are mathematically determined by the coordinates of the
  screen pixel and the look-at direction.  Since this is an orthogonal
  projection, all rays go in the same direction.  The actual picture
  of the loop drawn on the screen is used only for pixel region determination,
  it is not used for coloring the synthetic xray image - that is all done
  with the mathematical descriptions of the cylinders.

 2
  The openGL projection matrix used to map the current cylinder to the
  screen can be inverted to map the ray from screen coordinates to
  cylinder coordinates (where the mathematics for the cylinder are
  simpler).

 3
  Determining which screen pixels contain information from the cylinder
```

```
    can be done by drawing a rectangle around the cylinder in screen
    coordinates.  All pixels inside this rectangle will be used to start
    rays into the scene.  Each of these rays will be transformed using the
    inverse of the projection matrix and then intersected with the cylinder.

4
    If the ray does intersect the cylinder, (The intersection must also be
    between the two cylinder end planes.) the distance between the two
    intersections will be calculated.

    The distance is further refined by a density factor and a temperature
    factor.  So distances through regions where the density and/or
    temperature are low will not be as bright as regions where the
    temperature and density are relatively high.

5
    The depth of penetration information will be converted to an appropriate
    value and added to the current pixel value.  Thus, if more than one
    cylinder is penetrated by the same ray, the sum of the distances through
    all cylinders will determine the pixel color.

*****************************************************************************/

//Define whichever one of these is appropriate for the system.
//This define needs to be the same for all the *.cpp files and
//the sxidata.h file.

#include <fstream.h>
#include <process.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#ifdef WIN32
#include <afxwin.h>
#include <wingdi.h>
#endif

#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glaux.h>

#include "sxi.h"

//GLOBALS
GLubyte xrayData[HEIGHT][WIDTH];
GLdouble initInverse[ DIM * DIM ];
ifstream inFile;
ofstream imageOut;

// Screen pixels.  NOTE this size must be the same as the dimensions of
// xrayData[][] or pixel effects WILL be seen.
```

```
int viewport[] = {0, 0, 600, 600};

// Colormaps
float redmap[256], greenmap[256], bluemap[256];

// Viewing transformation
GLdouble EYEX = (GLdouble)0.2;
GLdouble EYEY = (GLdouble)-0.5;
GLdouble EYEZ = (GLdouble)0.1;
GLdouble CENTERX = (GLdouble)0.0;
GLdouble CENTERY = (GLdouble)0.0;
GLdouble CENTERZ = (GLdouble)0.0;
GLdouble UPX = (GLdouble)0.0;
GLdouble UPY = (GLdouble)0.0;
GLdouble UPZ = (GLdouble)1.0;

// Scene dimensions
GLfloat xMin = (GLfloat)-3.0;
GLfloat xMax = (GLfloat)3.0;
GLfloat yMin = (GLfloat)-3.0;
GLfloat yMax = (GLfloat)3.0;
GLfloat zMin = (GLfloat)-3.0;
GLfloat zMax = (GLfloat)3.0;

// Mapping parameters
int sensFcn = 0;
GLdouble exposure = 1.0;
char pixelType[20] = "MAKE_IMAGE_FIT";

int main( int argc, char *argv[] )
{
        int statusOK = SUCCESS;

 if( argc != 2 )
 {
  cerr << "Usage:  SXI <datafile name>" << endl;
  exit( NO_INPUT_FILE_NAME );
 }
/*
 //This may ultimately be used for drawing some kind of background
        if( argc == 1 )
          statusOK = createData( xrayData );
        else
        {
          infile->open( argv[1] );
          statusOK = infile->fail();
          if( statusOK == FAILURE )
          {
                  cout << "Data not available." << endl;
                  exit(1);
          }
          if( statusOK == SUCCESS )
```

```
                statusOK = readData( xrayData, infile );
        }*/



        if( statusOK == SUCCESS )
        {
          myinit( argv );
          auxReshapeFunc( myReshape );
          auxMainLoop( display );
          if(glGetError())
                cout << glGetError() << endl;
        }
        return( statusOK );
}

//Used to initialize some opengl stuff.
void myinit( char *argv[] )
{
 ifstream inCFile( "sxi.ini", ios::nocreate );
        int i;
 char outImage[80];
 GLfloat size;

 // Open the input data file and output image file.
 inFile.open( argv[1], ios::nocreate );
 if( !inFile )
 {
  cerr << "Unable to open " << argv[1] << ", exiting" << endl;
  exit( NO_DATA_FILE );
 }

 strcpy( outImage, argv[1] );
 strcat( outImage, ".bmp" );
 imageOut.open( outImage, ios::binary|ios::out );
 if( !imageOut )
 {
  cerr << "Unable to open " << argv[1] << ".bmp, exiting" << endl;
  exit( NO_BMP_FILE );
 }

 // Put a light up high.
 GLfloat lt0_pos[] = {10.0f,40.0f,100.0f, 0.0f};

 // Make a substantial glow from all directions.
 GLfloat lt0_amb[] = {0.6f,0.6f,0.6f, 1.0f};

 //Read in the configuration values.
 if ( !inCFile )
 {
  cerr << "Unable to open 'sxicfg.in' - continuing by using" << endl;
  cerr << "default values for configuration variables." << endl;
```

```
}
else
{
 inCFile >> viewport[0] >> viewport[1] >> viewport[2] >> viewport[3];
 inCFile >> EYEX >> EYEY >> EYEZ >> CENTERX >> CENTERY >> CENTERZ >>
        UPX >> UPY >> UPZ;
 inCFile >> sensFcn >> exposure >> pixelType >> size;
 pixelType[19] = '\0';
 if ( strcmp( pixelType, "FIXED_AREA_SIZE" ) )
 {
  // Is FIXED_PIXEL_SIZE so calculate area.
  xMax = size * WIDTH / (float)2;
 }
 else
 {
  // Is FIXED_AREA_SIZE so set up one boundary.
  xMax = size / (float)2;
 }
 xMin = -xMax;
 yMin = xMin;
 yMax = xMax;
 zMin = xMin;
 zMax = xMax;

 if ( inCFile.bad() )
 {
  cerr << "Unable to read in all configuration values." << endl;
  cerr << "Some variables may use default values, or even ";
  cerr << "unsuitable values." << endl;
 }
 inCFile.close();
}

      auxInitDisplayMode( AUX_SINGLE | AUX_RGB );

/* Set initial window position on screen. */
      //auxInitPosition (lowerLeftX, lowerLeftY, upperRightX, upperRightY);
      auxInitPosition ( viewport[0], viewport[1], viewport[2], viewport[3] );
      auxInitWindow (argv[0]); /* Opens a window on the screen. */
      for(i = 0; i < 256; i++)
      {
 //These values must be betweeen 0 and 1
 //SAVE THIS SET - the colors look pretty good.
        //redmap[i] = (float)(150 + i/3)/256;
        //greenmap[i] = (float)i/256;
        //bluemap[i] = (float)0.0;
        redmap[i] = (float)(150.0 + i/3.0)/(float)256.0;
        greenmap[i] = (float)i/(float)256.0;
        bluemap[i] = (float)0.1;
      }
redmap[0] = (float)0.0;
greenmap[0] = (float)0.0;
```

```
 bluemap[0] = (float)0.0;
        glPixelMapfv(GL_PIXEL_MAP_I_TO_R, 256, redmap);
        glPixelMapfv(GL_PIXEL_MAP_I_TO_G, 256, greenmap);
        glPixelMapfv(GL_PIXEL_MAP_I_TO_B, 256, bluemap);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

        glShadeModel (GL_FLAT);
        glEnable(GL_COLOR_MATERIAL);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glLightfv(GL_LIGHT0, GL_POSITION, lt0_pos );
 glLightfv(GL_LIGHT0, GL_AMBIENT, lt0_amb );

 cylQuadric = gluNewQuadric(); //Initialize the quadric for use in
                    //making cylinders and using their parameters.

}

/*****************************************************************************
 NOTE: matrix is 4 x 4 array of floats or doubles stored in column-major order
 Thus second element in array is row 2 column 1 in matrix. Can use glGet with
 params: (GL_PROJECTION_MATRIX, &array) to get current matrix value.

 This program intends to set up a view of a given section of the Sun (say
 600,000km across, and 300,000km high). The data in the configuration file
 will give the size of its view area. This size will be used to scale all
 data values before they enter the program. Currently this scaling is not
 implemented.

 The internal representation in the program will be of a volume that is 6
 units across (X and Y) and 3 units high (Z). Data points that are outside
 of this area will be flagged and printed out. This range checking will be
 done as the data are read the first time.
*****************************************************************************/

/*****************************************************************************
 Called when the window is first opened and whenever the window is
 reconfigured.
*****************************************************************************/
void CALLBACK myReshape( int w, int h )
{
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();          /* initialize the matrix */
 GLfloat xOrthoMin, xOrthoMax, yOrthoMin, yOrthoMax, zOrthoMin, zOrthoMax;
 GLfloat adj;

 //Scale the viewing volume so the entire x-y plane will fit. When viewing
 //at an angle to the x-axis (or y-axis) that is not 90 or 0 degrees, then
 //the viewing volume needs to be expanded so that the corners of the x-y
 //plane do not get cut off.
 adj = (GLfloat)1.0;
 if( ( EYEX != (GLfloat)0.0 ) || ( EYEY != (GLfloat)0.0 ) )
```

```
 {
  if( fabs( EYEX ) > fabs( EYEY ) )
  {
   adj = (GLfloat)( 1.0 + ( 0.4 * fabs( EYEY /  EYEX ) ) );
  }
  else
  {
   adj = (GLfloat)( 1.0 + ( 0.4 * fabs( EYEX /  EYEY ) ) );
  }
 }
 if( EYEZ != (GLfloat)0.0 )
 {
  adj = adj + (GLfloat)fabs( EYEZ / zMax );
 }

 xOrthoMin = adj * xMin;
 xOrthoMax = adj * xMax;
 yOrthoMin = adj * yMin;
 yOrthoMax = adj * yMax;
 zOrthoMin = adj * zMin;
 zOrthoMax = adj * zMax;

 //define the viewing volume
 glOrtho( xOrthoMin, xOrthoMax, yOrthoMin, yOrthoMax, zOrthoMin, zOrthoMax );
 /*------------------------------------------------------------------------
   Origin is at middle of screen if xMin = -xMax and yMin = -yMax.
   Ortho view mode makes a cut cube from the 6 planes defined in x,y,z
   min/max order.  Viewing is from the negative z axis.  This can be
   modified by using the gluLookAt function.

   Because this is a parallel projection system, the distance from the
   viewer to the viewing volume is immaterial.  All that matters is mapping
   the volume to screen coordinates, and setting the direction of view.
 ------------------------------------------------------------------------*/
 glMatrixMode( GL_MODELVIEW );
 glLoadIdentity();
       glClearColor( 0.5f, 0.5f, 0.5f, 0.0f ); // set background color
       glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
 gluLookAt( EYEX, EYEY, EYEZ, CENTERX, CENTERY, CENTERZ, UPX, UPY, UPZ );
}

/*********************************************************************
* Perhaps, rather than making a string of cylinders - in which the matching
* of the ends is rather difficult and unavoidably contains some error,
* it would be possible to use a nurbs surface created from the necklace
* of points.  This surface would probably need some tweaking to get
* it close enough to the points, but it might work quite well.  It certainly
* would eliminate cylinder end-point problems.  This idea will not be
* pursued for now.
*********************************************************************/

/*****************************************************************************
```

```
 Called when the window is first opened and whenever the window is
 reconfigured.
*****************************************************************************/
void CALLBACK display( )
{
        int statusOK, i, j;
 GLdouble initMat[ DIM * DIM ];
        int dataCheck;
//ifstream inFile( "xdata.in", ios::nocreate );

        glClear(GL_COLOR_BUFFER_BIT);   /* Takes care of background painting.*/

        // Initialize xrayData[][] to black
for( i = 0; i < HEIGHT; i++ )
        {
          for( j = 0; j < WIDTH; j++ )
          {
                  xrayData[i][j] = (GLubyte)0;
          }
}

// Place a color table in bottom left of pixel array xrayData[][]
for( i = 0; i < 20; i++ )
{
 for( j = 0; j < 256; j++ )
 {
  xrayData[i][j] = (GLubyte)j;
 }
}

SetupAxes();

glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );

//Get the current MODELVIEW matrix and store it's inverse for later use.
glGetDoublev( GL_MODELVIEW_MATRIX, initMat );
InvertMatrix( initMat, initInverse );

/*---------------------------------------------------------------------
 This section loops as long as there are still records in the input file.
 The first three records (enough to set up the following loop)
 were read in the init() function.  Now we repeatedly move each
 disk over one and read in one new record.  Using the four records (disks)
 we construct one cylinder.  At the end of the file, we are done.

 Here we read in a line from the input data file as a string.
 Then check that line to see if it is valid data or if it is the
 END_OF_LOOP code.  (Or some other code).  If the line is valid, process
 it as part of the current plasma loop.  If the line is the END_OF_LOOP
 code, then prepare to render another plasma loop.  At end of file, save
 resulting picture in a suitable data file - preferably gif format.
 FUTURE: An optional title may also be entered in the data file.  The
```

```
  title is on the line that begins with the code TITLE.
-------------------------------------------------------------------------------*/

dataCheck = GOOD_DATA;
while ( dataCheck != END_OF_FILE )
{
  dataCheck = MakePlasmaLoop( inFile );
}
inFile.close();

glPopMatrix();  // end of make plasma loops section

// Display the xray image pixel buffer.
statusOK = displayData( xrayData );

// Save the synthetic xray image as a BMP file
SaveImage( xrayData, imageOut );

      glFlush();
}
```

APPENDIX B

CYLINDER.CPP

```
/*==========================================================================

  cylinder.cpp    Steve Lundberg     10/10/97

  This file contains functions for drawing a tapered cylinder.

==========================================================================*/
#include <fstream.h>
#include <process.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#ifdef WIN32
#include <afxwin.h>
#include <wingdi.h>
#endif

#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glaux.h>

#include "sxi.h"

// GLOBAL VARIABLES
GLUquadricObj *cylQuadric; //Used for the construction of each cylinder.

/*****************************************************************************
  MakeCylinder() takes two DISKs containing the center position and
  radius at the bottom and top of the cylinder.  It also takes two
  more DISKs containing the prior and after points and radii.
  It makes a tapered cylinder using the DISKs for bottom and top of
  the cylinder.  Then the cylinder is lengthened and trimmed using a
  cut plane calculated from the two points that surround the point
  used for each end.  This cut plane is defined as the plane
  perpendicular to the line connecting the two surrounding points
  and containing the end point.

  MakeCylinder() also calls MapCylinderPixels() to place depth info
  about this cylinder in the pixel array.  This needs to be done at
  the time that the complete projection matrix for this cylinder is
  available.
*****************************************************************************/
void MakeCylinder( DISK prior, DISK bot, DISK top, DISK after )
{
  PLANE cutPl1, cutPl2, cylVect;
  GLfloat angZ1, angZ2, deltaZBot = 0.0f, deltaZTop = 0.0f, dot;
  DISK newTop;
  GLdouble eqn1[4];
  GLdouble eqn2[4];
  GLdouble density;
```

```
glColor3f( 0.6f, 0.0f, 0.4f ); //shade of violet

glPushMatrix(); //Save matrix position.
/*-------------------------------------------------------------------------
 Calculate the angle between the cut planes and the axis of the cylinder.
 If the plane is perpendicular to the axis of the cylinder, angZ is 0 deg.

 cos( angle ) = u.v/|u||v|  where u and v are vectors.  u = plane normal,
 v = cylinder axis.  Thus angle = arccos( u.v/|u||v| ) in radians.  Use
 the absolute value of the dot product so that angle is always positive.
-------------------------------------------------------------------------*/
//Cylinder vector (centerline).
cylVect.xParam = bot.center.xVal - top.center.xVal;
cylVect.yParam = bot.center.yVal - top.center.yVal;
cylVect.zParam = bot.center.zVal - top.center.zVal;
cylVect.offset = 0.0;
//Define the bottom cut plane.
CutPlane( prior.center, bot.center, top.center, &cutPl1 );
dot = DotProd( cylVect, cutPl1 );
angZ1 = RadToDeg * (float)acos( fabs( dot ) );
CheckAngle( angZ1, bot.center ); //Make sure angle is within limits

//Calculate cut plane for top.
CutPlane( bot.center, top.center, after.center, &cutPl2 );
dot = DotProd( cylVect, cutPl2 );
angZ2 = RadToDeg * (float)acos( fabs( dot ) );
CheckAngle( angZ2, top.center ); //Make sure angle is within limits

SetEqn( cutPl1, eqn1, 1 );
glClipPlane( GL_CLIP_PLANE0, eqn1 ); //clip for bottom end of cylinder
glEnable( GL_CLIP_PLANE0 );

SetEqn( cutPl2, eqn2, -1 );
glClipPlane( GL_CLIP_PLANE1, eqn2 ); //clip for top end of cylinder

/*-------------------------------------------------------------------------
 Clip planes are in eye coordinates, not scene coordinates, so they will
 be a rendered a little bit off what they really are.  The mathematical
 equations will be correct and the resulting x-ray image will be correctly
 calculated.
-------------------------------------------------------------------------*/
glEnable( GL_CLIP_PLANE1 );

/*-------------------------------------------------------------------------
 Position the cylinder.  Translate so that bottom end is at origin.
 Rotate so that centerline is on Z-axis.  Calculate bottom cut
 plane.  Raise above Z-axis by enough that the angled cut plane
 will not go below XY plane before it finishes cutting the end.
 After cutting the end, put cylinder back so that the center of
 the bottom is at origin.  Calculate top cut plane.  At top,
 extend top point up Z-axis by enough that top cut plane will
 cut entirely through cylinder.
```

```
-------------------------------------------------------------------------*/
MoveToOrigin( bot.center );
AlignZAxis( bot.center, top.center );
//Now the cylinder we want to build is aligned along the positive Z-axis
//and its bottom is at the origin.

//Move cylinder up above origin - along Z-axis - enough to allow cut.
ExtendCyl( angZ1, bot, &deltaZBot );  //How much to move
//Now move origin down so cylinder will be longer by deltaZBot
glTranslatef( (GLfloat)0.0, (GLfloat)0.0, -deltaZBot );

//Extend top of cylinder.
ExtendCyl( angZ2, top, &deltaZTop );  //How much to extend
newTop.center.zVal = DistBetweenPoints( top.center, bot.center ) +
                deltaZTop + deltaZBot;
newTop.center.xVal = top.center.xVal;
newTop.center.yVal = top.center.yVal;
newTop.radius = top.radius;

//Create the cylinder - extended so that the cuts can be made
gluCylinder( cylQuadric, bot.radius, top.radius, newTop.center.zVal, 10, 10 );

// Turn off clip planes so they don't affect any other objects.
glDisable( GL_CLIP_PLANE0 );
glDisable( GL_CLIP_PLANE1 );

density = ( top.density + bot.density ) / 2.0;
//Now put ray data in pixel array.
MapCylinderPixels( bot.radius, top.radius, newTop.center.zVal,
              cutPl1, cutPl2, density, bot.temperature, top.temperature );
glPopMatrix();  //Restore matrix - put cylinder back where it belongs

      glBegin(GL_LINES); // Draw centerline of cylinder
        glColor3f( 0.0f,1.0f,1.0f);
  glVertex3d( bot.center.xVal, bot.center.yVal, bot.center.zVal );
  glVertex3d( top.center.xVal, top.center.yVal, top.center.zVal );
 glEnd();

}

/*************************************************************************
 Move this point of the object to the origin.
*************************************************************************/
void MoveToOrigin( POINT3 pt )
{
 glTranslated( pt.xVal, pt.yVal, pt.zVal );
}

/*************************************************************************
 Calculate rotation vector and rotation angle to bring centerline to Z-axis.
*************************************************************************/
void AlignZAxis( POINT3 botPt, POINT3 topPt )
```

```
{
 POINT3 pt;
 GLfloat ang;

 //Set pt = end of vector from origin parallel to centerline
 pt.xVal = topPt.xVal - botPt.xVal;
 pt.yVal = topPt.yVal - botPt.yVal;
 pt.zVal = topPt.zVal - botPt.zVal;

 //Calculate # degrees to rotate = angle between vector from
 //origin to pt and the Z-axis.
 //length = sqroot of (pt.xVal**2 + pt.yVal**2 + pt.zVal**2)
 //angle = arccos( z/length )

 ang = (GLfloat)( RadToDeg * acos( pt.zVal /
     sqrt( pow( pt.xVal, 2 ) + pow( pt.yVal, 2 ) + pow( pt.zVal, 2 ) ) ) );

 glRotatef( -ang, (GLfloat)pt.yVal, -(GLfloat)pt.xVal, (GLfloat)0.0 );
}

/*****************************************************************************
 Return plane through middle point and bisecting the angle made by the
 lines from the left and right points.  Planar form: Ax + By + Cz + D = 0.
*****************************************************************************/
void CutPlane( POINT3 left, POINT3 middle, POINT3 right, PLANE * pl )
{
 float  lenright, lenleft;

 /*-------------------------------------------------------------------
  Angle bisector is the plane perpendicular to line from left to right
  only if the length from the middle to each of the other points is
  the same.  So, first find a new right point that is on original vector
  to right point, but is exactly as far away as the left point is from
  the middle.
 -------------------------------------------------------------------*/

 lenright = (float)sqrt( pow( right.xVal - middle.xVal, 2 ) +
                  pow( right.yVal - middle.yVal, 2 ) +
                  pow( right.zVal - middle.zVal, 2 ) );
 lenleft = (float)sqrt( pow( left.xVal - middle.xVal, 2 ) +
                 pow( left.yVal - middle.yVal, 2 ) +
                 pow( left.zVal - middle.zVal, 2 ) );

 /*-------------------------------------------------------------------
  Now construct vector from new right to original left points.  This is
  The normal to a plane that is perpendicular to the vector.  The equation
  for the vector from new right to original left point is: D - A.
  Where D is: end of vector from B in direction of original C-B vector
  and of same length as vector from A to B, or: D = B + new right vector.
  The new right vector starts at B, goes in the direction (C-B), and is
  adjusted to have same length as vector from A to B: |B-A|/|C-B|
  Thus D is: B + (C-B)|B-A|/|C-B|.  This yields the equation:
```

```
   D - A = (B + (C-B)|B-A|/|C-B|) - A.  Where A is left point, B is middle
   point, C is original right point, D is new right point, |B-A| is length
   of left vector, and |C-B| is length of original right vector.
   This equation holds for each of the three directions.
  ----------------------------------------------------------------------*/

  pl -> xParam = ( middle.xVal +
             ( right.xVal - middle.xVal ) * lenleft / lenright ) -
              left.xVal;
  pl -> yParam = ( middle.yVal +
             ( right.yVal - middle.yVal ) * lenleft / lenright ) -
              left.yVal;
  pl -> zParam = ( middle.zVal +
             ( right.zVal - middle.zVal ) * lenleft / lenright ) -
              left.zVal;

  //The plane contains the middle point
  pl -> offset = -1 * ( pl -> xParam * middle.xVal + pl -> yParam * middle.yVal +
          pl -> zParam * middle.zVal );
}

/*****************************************************************************
  Return amount to extend cylinder along Z-axis to allow room for cut plane
  to cut entirely through outside wall of cylinder.  The cut plane must not
  intersect the new end plane of the cylinder except at one point.

  This extension will be at constant radius so that both cylinders that meet
  at this point will have the same cross section where they meet at the cut
  plane.  This introduces some error in the case where the cylinders are
  actually truncated cones.  The effect of the error will be to make the
  cylinder which has its small end at this junction a little larger than it
  was supposed to be, and to make the cylinder which has its large end at
  this junction a little smaller than it was supposed to be.  Larger bends
  will lead to larger errors, so a note will be displayed if there is more
  than 10 degrees bend at any junction.
*****************************************************************************/
void ExtendCyl( GLfloat angZ, DISK thisEnd, GLfloat * deltaZ )
{
  angZ = angZ / RadToDeg ;  //Convert angZ back to radians
  //angZ must be <= 45 deg.
  *deltaZ = (GLfloat)thisEnd.radius * (GLfloat)tan( angZ );
}

/*****************************************************************************
  Check the angle between two cylinders.  If it is greater than 45deg. exit
  the program with an error.  If it is larger than 10deg. display a warning.
*****************************************************************************/
void CheckAngle( GLfloat angZ, POINT3 pt )
{
  if ( angZ > 45.0 ) //For flat end, the cut plane angle is 0 deg.
  {
    cerr << " ERROR: the angle between cylinders is too great." << endl;
```

```
    cerr << "  The angle is: " << angZ << " degrees." << endl;
    cerr << "  The center point is at (x,y,z): "
         << pt.xVal << ", " << pt.yVal << ", " << pt.zVal << endl;
    exit( LARGE_ANGLE );
   }

   if ( angZ > 10.0 ) //Leads to larger errors - notify user
   {
    cout << " NOTE: the angle between adjacent cylinders" << endl;
    cout << "  is greater than 10 degrees.  This may yield" << endl;
    cout << "  unacceptably large errors in the mapping process." << endl;
    cout << "  The center point is at (x,y,z): " << pt.xVal << ", " << pt.yVal
         << ", " << pt.zVal << endl;
    cout << "  The angle is: " << angZ << endl;
   }
}

/*****************************************************************************
  Calculate the dot product for two vectors defined by two planes.
*****************************************************************************/
GLfloat DotProd( PLANE pl1, PLANE pl2 )
{
  double numer, denom;
  GLfloat dot;

  numer = pl1.xParam * pl2.xParam +  pl1.yParam * pl2.yParam +
          pl1.zParam * pl2.zParam;
  denom = sqrt( pow( pl1.xParam, 2 ) + pow( pl1.yParam, 2 ) +
                pow( pl1.zParam, 2 ) ) *
          sqrt( pow( pl2.xParam, 2 ) + pow( pl2.yParam, 2 ) +
                pow( pl2.zParam, 2 ) );

  dot = (GLfloat)( numer / denom );
  return( dot );
}

/*****************************************************************************
  Make planar equation from PLANE - set 'up' direction.
*****************************************************************************/
void SetEqn( PLANE pl, GLdouble eqn[], GLint up )
{
  eqn[0] = (GLdouble)( up * pl.xParam );
  eqn[1] = (GLdouble)( up * pl.yParam );
  eqn[2] = (GLdouble)( up * pl.zParam );
  eqn[3] = (GLdouble)( up * pl.offset );
}

/*****************************************************************************
  Calculate the distance between two points.
*****************************************************************************/
GLfloat DistBetweenPoints( POINT3 pt1, POINT3 pt2 )
{
```

```
return ( (GLfloat)sqrt( pow( ( pt2.xVal - pt1.xVal ), 2 ) +
                pow( ( pt2.yVal - pt1.yVal ), 2 ) +
                pow( ( pt2.zVal - pt1.zVal ), 2 ) ) );
}
```

APPENDIX C

CYLMAP.CPP

```
/*=============================================================================

  cylmap.cpp      Steve Lundberg     7/20/97

  This file contains the functions used to perform ray-tracing
  from the screen image of a cylinder (in the position it is
  initially created - z-axis symetric, bottom on origin) through
  the cylinder.  The length of the ray through the cylinder is
  then used to scale the color of the pixel where the ray originated.

=============================================================================*/
#include <stdlib.h>
#include <fstream.h>
#include <process.h>
#include <math.h>

#ifdef WIN32
#include <afxwin.h>
#include <wingdi.h>
#endif

#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glaux.h>

#include "sxi.h"

//DEBUG GLOBAL
static int DEBUG_MOVE_POINTS = 0;
static int DEBUG_CLIP_PLANE = 0;

/*****************************************************************************
  First, calc. direction vector for screen into scene.  Since we are using
  parallel projection there is only one direction vector for the entire scene.
  Second, calc. bounding rectangle for cylinder in screen coords.
  Third, for each pixel in bounding rectangle, determine if it intersects
  with the cylinder.  This requires mapping the screen pixels and rays to
  the cylinder coordinates, then calculating the intersections.
  Fourth, if the cylinder intersection is outside the clip planes for the
  ends of this cylinder, replace this intersection with the intersection of
  the ray with the clip plane.  But, if the second ray intersection with the
  cylinder is also outside the same clip plane, then this ray doesn't intersect
  the clipped cylinder at all.
  Fifth, calculate the distance between the two points of intersection.
  Sixth, set the xray image buffer pixel color proportional to the distance.
  This function is called with the modelMatrix for the construction of the
  cylinder - i.e. cylinder axis is z-axis, bottom of cylinder at z = 0.
*****************************************************************************/
void MapCylinderPixels( GLdouble botRad, GLdouble topRad, GLdouble height,
                        PLANE cutPl1, PLANE cutPl2, GLdouble density,
                        GLdouble sensitivity1, GLdouble sensitivity2 )
{
```

```
int init;
PLANE scrPlane;
POINT3 cylPt, scrPt;
int i, j;
GLdouble modelMat[ DIM * DIM ];
GLdouble projMat[ DIM * DIM ];
GLdouble modelMatFixed[ DIM * DIM ];
GLdouble dist;
int counts;

GLdouble minMax[ 4 ]; // index 0 is x-min, 1 is x-max, 2 is y-min, 3 is y-max.
int xPos, yPos;

//Part 1: vector = eye pt - center pt.
scrPlane.xParam = (GLfloat)(EYEX - CENTERX);
scrPlane.yParam = (GLfloat)(EYEY - CENTERY);
scrPlane.zParam = (GLfloat)(EYEZ - CENTERZ);

/*----------------------------------------------------------------------
Part 2:
The bounding rect. for the cyl. is easily found in the coords. where
it is initially created. Using the un-clipped cyl. gives slightly larger
values than absolutely necessary, but is much easier. When the cyl. is
created, its bottom lies in the x-y plane (z = 0) and extends out a radius.
When viewed from any angle, the points x = y = +/- SQRT2*radius will always
be bounds for this circle. Similarly for the top. So if the eight points,
(4 for the bottom and 4 for the top) are transformed to screen coords. and
then used to generate min/max values, the resulting rect. bounds the
cylinder on the screen.

Using gluProject() will give the x and y values for this rectangle in pixel
numbers from the origin of the screen (lower left corner).

Calculate the transform of one of the eight points, then update values of
min and max in screen coords. Loop over all eight points.
----------------------------------------------------------------------*/
glGetDoublev( GL_MODELVIEW_MATRIX, modelMat );
glGetDoublev( GL_PROJECTION_MATRIX, projMat );

MatrixMult( initInverse, modelMat, modelMatFixed );

init = TRUE; //Used to initialize minMax[]

// Bottom points: z = 0
cylPt.zVal = (GLfloat)0.0;

glColor3f(0.0f,0.0f,1.0f);//blue

for ( i = -1; i < 2; i+=2 )
{
 cylPt.xVal = i * SQRT2 * botRad;
 for ( j = -1; j < 2; j+=2 )
```

```
  {
   cylPt.yVal = j * SQRT2 * botRad;
   gluProject( cylPt.xVal, cylPt.yVal, cylPt.zVal,
    modelMat, projMat, viewport,
    &scrPt.xVal, &scrPt.yVal, &scrPt.zVal );

   MinMax( minMax, scrPt, &init );   // Update x and y min/max values
  }
 }


// Top points: z = height
cylPt.zVal = height;
for ( i = -1; i < 2; i+=2 )
{
 cylPt.xVal = i * SQRT2 * topRad;
 for ( j = -1; j < 2; j+=2 )
 {
  cylPt.yVal = j * SQRT2 * topRad;
  gluProject( cylPt.xVal, cylPt.yVal, cylPt.zVal,
   modelMat, projMat, viewport,
   &scrPt.xVal, &scrPt.yVal, &scrPt.zVal );

  MinMax( minMax, scrPt, &init );   // Update x and y min/max values
 }
}

/*-------------------------------------------------------------------------
 Part 3:
  Part 4 and Part 5 included in this section.
  Now minMax[] contains points representing the bounding rect. in screen
  pixel coordinates.
 Part 6:  Set the color for the screen pixels.
 This will be done using an additive process so that if
 other cylinder points are mapped to this pixel, their contribution
 to the total can be added in.  The temperature is scaled proportional to
 the distance along the path between the cylinder end points.  An exposure
 time factor is also used to allow for enhancing different sections of the
 image.

 This section is heavily openGL dependent whereas most of the preceeding
 work is not.

 The distance values are scaled to a color value and written to a
 pre-defined storage buffer.

 xPos and yPos are pixel numbers.
-------------------------------------------------------------------------*/
// First make sure the bounding rectangle is entirely within the displayable
// image.  Trim it to fit if it isn't.
if ( minMax[0] < 0 )
{
```

```
  minMax[0] = 0;
 }
 if ( minMax[2] < 0 )
 {
  minMax[2] = 0;
 }
 if ( minMax[1] > WIDTH )
 {
  minMax[1] = WIDTH;
 }
 if ( minMax[3] > HEIGHT )
 {
  minMax[3] = HEIGHT;
 }

 for ( yPos = (int)minMax[2]; yPos < (int)minMax[3]; yPos++ )
 {
  for ( xPos = (int)minMax[0]; xPos < (int)minMax[1]; xPos++ )
  {
   dist = CalculateIntersectionDistance( xPos, yPos,
              projMat, modelMat, modelMatFixed, botRad, topRad, height,
              cutPl1, cutPl2 );

   // Scale distance by density and sensor sensitivity.
   // The distance is 1.0 for an intersection distance = 1/6 of horizontal span.
   // Density is the square of the electron number density (read from the input file).
   // The exposure variable allows for brightness adjustment.
   // The result is divided by 16 to make the count match what would be
   // stored in a 12-bit value.  (This value is 8 bits.)
   counts = (int)( dist * ( ( xMax - xMin ) / 6 ) * density *
          ( ( sensitivity1 + sensitivity2 ) / 2 ) *
          exposure / 16 );
   AdjustPixelColor( xPos, yPos, counts );
  }
 }
}

/*****************************************************************************
 Function to find the min and max of several bounding points.
*****************************************************************************/
void MinMax( GLdouble minMax[], POINT3 scrPt, int* init )
{
 if ( *init == TRUE )
 {
  minMax[ 0 ] = scrPt.xVal;
  minMax[ 1 ] = scrPt.xVal;
  minMax[ 2 ] = scrPt.yVal;
  minMax[ 3 ] = scrPt.yVal;
  *init = FALSE;
 }
 else
 {
```

```
    if ( scrPt.xVal < minMax[ 0 ] )
      minMax[ 0 ] = scrPt.xVal;
    else if ( scrPt.xVal > minMax[ 1 ] )
      minMax[ 1 ] = scrPt.xVal;

    if ( scrPt.yVal < minMax[ 2 ] )
      minMax[ 2 ] = scrPt.yVal;
    else if ( scrPt.yVal > minMax[ 3 ] )
      minMax[ 3 ] = scrPt.yVal;
  }
}

/*****************************************************************************
  Part 3 and Part 4:
  See if ray from this pixel intersects cylinder.
  Make sure that ray intersects twice, or else is not used.
  Part 5:
  If ray intersects cylinder, determine the distance between in and out points.

  Equations for right truncated cone.
  From z = 0 to z = z0
  At z = 0  have circle:   x*x + y*y = r1*r1
  At z = z0 have circle:   x*x + y*y = r2*r2
  circle at any z is:  x*x + y*y = (r1 - (r1-r2)(z/zr2))**2
  or:  x*x + y*y -(r1 - (r1-r2)(z/zr2))**2 = 0
  or:  x*x + y*y -(r1*r1 -2r1*(r1-r2)(z/zr2) + (r1-r2)**2(z/zr2)**2) = 0
  or:  x*x + y*y -r1*r1 +2r1*(r1-r2)(z/zr2) - (r1-r2)**2(z/zr2)**2) = 0
  coeff x*x:  1                  so: a(Q11) = 1
  coeff y*y:  1                  so: e(Q22) = 1
  coeff z*z:  -((r1-r2)/zr2)**2     so: h(Q33) = -((r1-r2)/zr2)**2
  coeff z:    2r1*(r1-r2)/zr2       so: i(Q34) = r1(r1-r2)/zr2
  const:      -r1*r1                so: j(Q44) = -r1*r1
  where zr2 is z value where radius is r2 and at z = 0 the radius is r1.
  All this can most easily be done in the cylinder coordinate system.
  The screen ray is transformed to cylinder coords, the intersections found,
  and then the distance between the points is found.
*****************************************************************************/
GLdouble CalculateIntersectionDistance( int xPos, int yPos,
                    GLdouble projMat[DIM * DIM],
                    GLdouble modelMat[DIM * DIM],
                    GLdouble modelMatFixed[DIM * DIM],
                    GLdouble botRad, GLdouble topRad,
                    GLdouble height, PLANE cutPl1, PLANE cutPl2 )
{
  // quadricData needs only three values, since all the other 13 values
  // of the quadric for a right truncated cylinder are either zero or one and
  // do not change.
  GLdouble quadricData[3];
  GLdouble dist = 0.0f;
  GLdouble lenVect;
  POINT3 scrPt, cylStartPt, cylVect, cylEndPt;
  POINT3 posRoot, negRoot;
```

```
POINT3 tempRoot;
RAY    cylRay;
int    numberRoots=0;

// Make ray from this pixel.  The ray must be in cylinder coordinates,
// but minMax[] and pixel position are in screen coordinates.  Use
// z = near and far clipping plane values (0.0 and 1.0).

// Transform point on screen to point in cylinder coordinates.
scrPt.xVal = (GLdouble)xPos;
scrPt.yVal = (GLdouble)yPos;
scrPt.zVal = 0.0;  //This is for the near clipping plane

//Find the start point of this ray in cylinder coordinates.
gluUnProject( scrPt.xVal, scrPt.yVal, scrPt.zVal,
          modelMat, projMat, viewport,
          &cylStartPt.xVal, &cylStartPt.yVal, &cylStartPt.zVal );

 scrPt.zVal = 2.0;  //This is for the far clipping plane
//Find the ending point of this ray in cylinder coordinates.
gluUnProject( scrPt.xVal, scrPt.yVal, scrPt.zVal,
          modelMat, projMat, viewport,
          &cylEndPt.xVal, &cylEndPt.yVal, &cylEndPt.zVal );

//Calculate the vector from the startPt to the endPt in cylinder coordinates.
cylVect.xVal = cylEndPt.xVal - cylStartPt.xVal;
cylVect.yVal = cylEndPt.yVal - cylStartPt.yVal;
cylVect.zVal = cylEndPt.zVal - cylStartPt.zVal;

//Make the direction vector have unit length.
lenVect = sqrt( pow( cylVect.xVal, 2 ) +  pow( cylVect.yVal, 2 ) +
        pow( cylVect.zVal, 2 ) );
cylVect.xVal = cylVect.xVal / lenVect;
cylVect.yVal = cylVect.yVal / lenVect;
cylVect.zVal = cylVect.zVal / lenVect;

cylRay.initPt = cylStartPt;
cylRay.dirVect = cylVect;
/*------------------------------------------------------------------------
Solve equation for roots - intersections.  Need equation for the
cylinder in question in cylinder coordinates - this is the quadric.
The equation is in the form of: k2*t*t + k1*t + k0 = 0.
For the case of a circular truncated cone (called cylinder for short):
k2 = ax*ax*Q11 + ay*ay*Q22 + az*az*Q33
k1 = 2*ax*x0*Q11 + 2*ay*y0*Q22 + 2*az(z0*Q33 + Q34)
k0 = x0*x0*Q11 + y0*y0*Q22 + z0*z0*Q33 + Q44 + 2*z0*Q34

Where:

  The Q's are the non-zero elements of a 4x4 matrix - the quadric
  form of the circular truncated cone.
```

ax, ay, and az are the direction vector elements for the ray.
x0, y0, and z0 are the starting end point values for the ray.

ref. Graphics Gems III p. 280

The general form for a quadric surface is:

$F(x,y,z) = ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fxy + 2gy + hz^2 + 2iz + j = 0$

The matrix form of a quadric is:

$$f(x,y,z) = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = XQX^t = 0$$

Q11 = a, Q12 = Q21 = b, Q14 = Q41 = d, etc.

For a right truncated cone with z the axis of symmetry the values of the non-zero elements in Q are: Q11 = Q22 = 1; Q33 = -(r1 - r2)**2/h**2; Q34 = Q43 = (r1**2 -r1r2)/h; and Q44 = -r1**2. Where r1 is the radius at z = 0, r2 is the radius at z = h, and h is the height of the cylinder.

NOTE: While it should be possible to derive the quadric for the cylinder with ends cut by arbitrary planes, since I have not done that, I will simply check all points to be sure they are not outside of the end planes. That means that some intersections will be with the end planes rather than with the cylinder itself.

Solve for the values of t using the quadratic equation. (Must first make sure that the portion in the square root is positive.) Must also make sure that the intersections lie both within the cylinder volume, and within the cut plane bounds of the cylinder.

```
t = (-b [+/-]SQRT(b*b - 4*a*c))/2*a   [a*x*x + b*x + c = 0]
-------------------------------------------------------------------------*/

// Initialize the quadric values for this cylinder.
// quadricData[0] = Q33 = -((r1 - r2)/h)**2
// quadricData[1] = Q34 = Q43 = (r1**2 -r1r2)/h
// quadricData[2] = Q44 = -r1**2
// Where r1 = botRad, r2 = topRad, and h = height
quadricData[0] = -( pow( ( botRad - topRad) / height, 2 ) );
quadricData[1] = ( pow( botRad, 2 ) - botRad * topRad ) / height;
quadricData[2] = - pow( botRad, 2 );

numberRoots = GetEndPoints( &posRoot, &negRoot, cylRay, quadricData );

// dist may still be zero if pts are not inside truncated cylinder.
// The cut planes are in display coords, so the points must be also.
if ( numberRoots == 2 )
{
```

```
  // Transform the intersection points from cylinder coords to scene coords.
  // This is needed because they need to be checked against the clip planes
  // and the clip plane equations are in scene coordinates.
  // Cannot use gluUnProject() and gluProject() here because those two
  // functions transform cylinder coords from/to screen coords, not scene
  // coordinates.
  tempRoot = TransformPoint( posRoot, modelMatFixed ); //From cylinder coords
  posRoot = tempRoot;          //To scene coords
  tempRoot = TransformPoint( negRoot, modelMatFixed ); //From cylinder coords
  negRoot = tempRoot;          //To scene coords

  if ( MovePointsWithinCutPlanes( &posRoot, &negRoot, cutPl1, cutPl2 ) )
  {
   dist = (GLdouble)DistBetweenPoints( posRoot, negRoot );
   if ( ( dist >= 0.8 ) && ( DEBUG_CLIP_PLANE ) )
   {
    cout << "large dist: " << dist << endl;
   }
  }
 }
 return ( dist );
}


/*****************************************************************************
First calculate the K-values, then set up the solutions using the
quadratic equation: (-b [+/-]SQRT(b**2 -4ac))/2a or
(-K1 [+/-]SQRT(K1**2 - 4*K2*K0))/2*K2.
*****************************************************************************/
int GetEndPoints( POINT3 * posRootPtr, POINT3 * negRootPtr, RAY cylRay,
          GLdouble * quadricData )
{
 GLdouble K0, K1, K2, detmnt, posRootT, negRootT;
 int numRoots = 0;

 K0 = pow( cylRay.initPt.xVal, 2 ) + pow( cylRay.initPt.yVal, 2 ) +
    pow( cylRay.initPt.zVal, 2 ) * quadricData[0] + quadricData[2] +
    2 * cylRay.initPt.zVal * quadricData[1];

 K1 = 2 * cylRay.dirVect.xVal * cylRay.initPt.xVal +
    2 * cylRay.dirVect.yVal * cylRay.initPt.yVal +
    2 * cylRay.dirVect.zVal *
         ( cylRay.initPt.zVal * quadricData[0] + quadricData[1] );

 K2 = pow( cylRay.dirVect.xVal, 2 ) + pow( cylRay.dirVect.yVal, 2 ) +
    pow( cylRay.dirVect.zVal, 2 ) * quadricData[0];

 detmnt = pow( K1, 2 ) - 4 * K2 * K0;
 if ( (detmnt >= 0) && (K0 != 0) )
 {
  if ( detmnt == 0 )
  {
```

```
    numRoots = 1; // tangential intersection: distance = 0.
   }
   else
   {
    numRoots = 2;
    posRootT = ( -K1 + sqrt( detmnt ) ) / ( 2 * K2 );
    negRootT = ( -K1 - sqrt( detmnt ) ) / ( 2 * K2 );
    posRootPtr -> xVal = cylRay.initPt.xVal +
                         (GLfloat)posRootT * cylRay.dirVect.xVal;
    posRootPtr -> yVal = cylRay.initPt.yVal +
                         (GLfloat)posRootT * cylRay.dirVect.yVal;
    posRootPtr -> zVal = cylRay.initPt.zVal +
                         (GLfloat)posRootT * cylRay.dirVect.zVal;
    negRootPtr -> xVal = cylRay.initPt.xVal +
                         (GLfloat)negRootT * cylRay.dirVect.xVal;
    negRootPtr -> yVal = cylRay.initPt.yVal +
                         (GLfloat)negRootT * cylRay.dirVect.yVal;
    negRootPtr -> zVal = cylRay.initPt.zVal +
                         (GLfloat)negRootT * cylRay.dirVect.zVal;
   }
  }

  return ( numRoots );
}

/*****************************************************************************
If poth points lie outside of the same cut plane, then they do not
represent a real intersection.  If only one lies outside, then that
ray needs to be intersected with the plane to find the true intersection
with the cut cylinder.  If both points lie outside of different cut planes
then both rays need to be intersected with their respective cut planes.
If neither lies outside the planes, then they are the correct intersection
points.
The cut planes are defined in scene coords.  The points need to be also.
*****************************************************************************/
int MovePointsWithinCutPlanes( POINT3 * posRootPtr, POINT3 * negRootPtr,
                    PLANE cutPl1, PLANE cutPl2 )
{
 int goodPts = TRUE;
 GLdouble pt1Pl1, pt1Pl2, pt2Pl1, pt2Pl2;
 RAY scrRay;

 // Substitute point value into planar eqn. to determine which side of the
 // plane the point is on.
 pt1Pl1 =  cutPl1.xParam * ( posRootPtr -> xVal ) +
        cutPl1.yParam * ( posRootPtr -> yVal ) +
        cutPl1.zParam * ( posRootPtr -> zVal ) +
        cutPl1.offset;
 pt1Pl2 =  cutPl2.xParam * ( posRootPtr -> xVal ) +
        cutPl2.yParam * ( posRootPtr -> yVal ) +
        cutPl2.zParam * ( posRootPtr -> zVal ) +
        cutPl2.offset;
```

```
pt2Pl1 =  cutPl1.xParam * ( negRootPtr -> xVal ) +
        cutPl1.yParam * ( negRootPtr -> yVal ) +
        cutPl1.zParam * ( negRootPtr -> zVal ) +
        cutPl1.offset;
pt2Pl2 =  cutPl2.xParam * ( negRootPtr -> xVal ) +
        cutPl2.yParam * ( negRootPtr -> yVal ) +
        cutPl2.zParam * ( negRootPtr -> zVal ) +
        cutPl2.offset;

// If both points lie outside of the same cut plane, then this ray does
// not intersect the cut cylinder.
if ( pt1Pl1 < 0 && pt2Pl1 < 0 ||
    pt1Pl2 > 0 && pt2Pl2 > 0 )
{
 goodPts = FALSE;
}
// Otherwise, be sure that both intersections are with portions of the
// cylinder that really exist after trimming by the cut planes.
// This is accomplished by moving any point that is outside the cut plane,
// but on the cylinder before it was cut, down the ray to the point where
// the ray intersects the plane on the end of the cylinder.
else
{
 goodPts = TRUE;
 //Create scrRay from the two points given as input.
 //It should not matter which way we point it (in or out).  The
 //calculated intersection with the plane will be the same either way.
 scrRay.initPt.xVal = posRootPtr -> xVal;
 scrRay.initPt.yVal = posRootPtr -> yVal;
 scrRay.initPt.zVal = posRootPtr -> zVal;
 scrRay.dirVect.xVal = negRootPtr -> xVal - posRootPtr -> xVal;
 scrRay.dirVect.yVal = negRootPtr -> yVal - posRootPtr -> yVal;
 scrRay.dirVect.zVal = negRootPtr -> zVal - posRootPtr -> zVal;

 if ( pt1Pl1 < 0 )
 {
  if ( DEBUG_CLIP_PLANE )
  {
   cout << "pt1Pl1" << endl;
   DEBUG_CLIP_PLANE = 0;
  }
  IntersectRayWithCutPlane( posRootPtr, cutPl1, scrRay );
 }
 else if ( pt1Pl2 > 0 )
 {
  if ( DEBUG_CLIP_PLANE )
  {
   cout << "pt1Pl2" << endl;
   DEBUG_CLIP_PLANE = 0;
  }
  IntersectRayWithCutPlane( posRootPtr, cutPl2, scrRay );
  DEBUG_CLIP_PLANE = 0;
```

```
    }

    if ( pt2Pl1 < 0 )
    {
     if ( DEBUG_CLIP_PLANE )
     {
      cout << "pt2Pl1" << endl;
      DEBUG_CLIP_PLANE = 0;
     }
     IntersectRayWithCutPlane( negRootPtr, cutPl1, scrRay );
    }
    else if ( pt2Pl2 > 0 )
    {
     if ( DEBUG_CLIP_PLANE )
     {
      cout << "pt2Pl2" << endl;
      DEBUG_CLIP_PLANE = 0;
     }
     IntersectRayWithCutPlane( negRootPtr, cutPl2, scrRay );
    }
   }

   return ( goodPts );
}

/*****************************************************************************
 Using planar eqn. Ax + By + Cz + d = 0 and a ray defined by a direction
 vector and an initial point with the ray written in parametric form as
 x = initPt.xVal + t(dirVect.xVal) and similarly for y and z, we can find
 the value of t at the point where the ray intersects the plane - if there
 is an intersection.  From Foley, VanDam, Feiner, Hughes, and Phillips
 p. 460-461:

 t = -(A*initPT.xVal + B*initPt.yVal + C*initPt.zVal + D) /
     (A*dirVect.xVal + B*dirVect.yVal + C*dirVect.zVal)

 If the denominator is 0, then the ray is parallel with the plane and there
 is no intersection.
*****************************************************************************/
void IntersectRayWithCutPlane( POINT3 * rootPtr, PLANE cutPl, RAY scrRay )
{
 GLdouble t, denom;

 denom = cutPl.xParam * scrRay.dirVect.xVal +
         cutPl.yParam * scrRay.dirVect.yVal +
         cutPl.zParam * scrRay.dirVect.zVal;

 if ( denom == 0.0 )  //Ray is parallel to plane and outside of cut cylinder.
 {                    //This should never happen since we have already checked.
  printf( "Ray did not intersect cut plane." );
  exit( IMAGINARY_INTERSECTION );
 }
```

```
  else
  {
   t = -( cutPl.xParam * scrRay.initPt.xVal +
        cutPl.yParam * scrRay.initPt.yVal +
        cutPl.zParam * scrRay.initPt.zVal + cutPl.offset ) / denom;

   rootPtr -> xVal = (GLfloat)(scrRay.initPt.xVal + scrRay.dirVect.xVal * t );
   rootPtr -> yVal = (GLfloat)(scrRay.initPt.yVal + scrRay.dirVect.yVal * t );
   rootPtr -> zVal = (GLfloat)(scrRay.initPt.zVal + scrRay.dirVect.zVal * t );
  }
 }

/*****************************************************************************
 Set color of designated pixel proportional to distance through cylinder.
 Dist is scaled before it is sent here.
 The incoming x and y positions are screen pixel numbers, they need to be
 scaled to fit the pixel buffer, which may not be the full screen size.
*****************************************************************************/
void AdjustPixelColor( int xPos, int yPos, int dist )
{
 GLubyte colorValue;

 //Negative distance shouldn't happen
 if ( dist < 0 )
 {
  dist = 0;
  cout << "Ray distance thru cylinder was negative. (x,y): " <<
   xPos << ", " << yPos << endl;
 }

 xPos = xPos * WIDTH / viewport[2];
 yPos = yPos * HEIGHT / viewport[3];

 if ( (yPos < HEIGHT) && (xPos < WIDTH) && (yPos >= 0) && (xPos >= 0) )
 {
  //accumulate into pixel image buffer.
  dist += (int)xrayData[ yPos ][ xPos ];

  //Saturation is 255
  if ( dist >= 255 )
  {
   dist = 255;
  }
  colorValue = (GLubyte)dist;
  xrayData[ yPos ][ xPos ] = colorValue;
 }
 else
 {
  cout << "Bad xrayData index y, x: " << yPos << ", " << xPos << endl;
 }
}
```

APPENDIX D

SXIUTILS.CPP

```
/*=============================================================================

  sxiUtils.cpp    Steve Lundberg    2/15/97

  Defines the function used to draw the synthetic xray picture.

  Defines the functions used to read in the data file, check the data
  for validity, and start the cylinder creation process.

  The function to output the bitmap file is here.

  For future possible use:
    Uses the rand() fcn to get numbers between 0 and 255.
    These values can be used to produce a surface plane to represent the
    surface of the sun.

=============================================================================*/
#include <stdlib.h>
#include <fstream.h>
#include <process.h>
#include <math.h>

#ifdef WIN32
#include <afxwin.h>
#include <wingdi.h>
#endif

#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glaux.h>

#include "sxi.h"

//Local prototypes
static double yohkoh_instr_rsp_00( double temp );
static double yohkoh_instr_rsp_01( double temp );
static double yohkoh_instr_rsp_02( double temp );
static double yohkoh_instr_rsp_03( double temp );
static double yohkoh_instr_rsp_04( double temp );
static double yohkoh_instr_rsp_05( double temp );

/*****************************************************************************
  Draws the pixel data in the array at the current screen location.
*****************************************************************************/
int displayData( GLubyte xrayData[HEIGHT][WIDTH] )
{
        glDrawPixels( WIDTH, HEIGHT, GL_COLOR_INDEX, GL_UNSIGNED_BYTE, xrayData );
        return( 0 );
}

void SetupAxes( void )
{
```

```
glPushMatrix();

glBegin( GL_LINES );  // Draw the three axes
        glColor3f( 1.0f, 0.0f, 0.0f );//red
        glVertex3d( xMin, 0, 0 );
        glVertex3d( xMax, 0, 0 );
        glColor3f( 0.0f, 1.0f, 0.0f );//green
        glVertex3d( 0, yMin, 0 );
        glVertex3d( 0, yMax, 0 );
        glColor3f( 0.0f, 0.0f, 1.0f );//blue
        glVertex3d( 0, 0, zMin );
        glVertex3d( 0, 0, zMax );
      glEnd();
glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );

// Draw 3 squares in the x-y plane
glBegin( GL_POLYGON );
        glColor3f( 0.0f, 0.0f, 1.0f );//blue
        glVertex3d( xMin/3, yMin/3, 0 );
        glVertex3d( xMin/3, yMax/3, 0 );
        glVertex3d( xMax/3, yMax/3, 0 );
        glVertex3d( xMax/3, yMin/3, 0 );
      glEnd();
glBegin( GL_POLYGON );
        glColor3f( 0.0f, 0.0f, 1.0f );//blue
        glVertex3d( 2 * xMin/3, 2 * yMin/3, 0 );
        glVertex3d( 2 * xMin/3, 2 * yMax/3, 0 );
        glVertex3d( 2 * xMax/3, 2 * yMax/3, 0 );
        glVertex3d( 2 * xMax/3, 2 * yMin/3, 0 );
      glEnd();
glBegin( GL_POLYGON );
        glColor3f( 0.0f, 0.0f, 1.0f );//blue
        glVertex3d( xMin, yMin, 0 );
        glVertex3d( xMin, xMax, 0 );
        glVertex3d( xMax, xMax, 0 );
        glVertex3d( xMax, yMin, 0 );
      glEnd();

//make a small sphere at x, y, and z = +3
glPushMatrix();
glTranslatef( xMax, 0.0f, 0.0f );
auxWireSphere( 0.02f );
glPopMatrix();
glPushMatrix();
glTranslatef( 0.0f, yMax, 0.0f );
auxWireSphere( 0.02f );
glPopMatrix();
glPushMatrix();
glTranslatef( 0.0f, 0.0f, zMax );
auxWireSphere( 0.02f );
glPopMatrix();
}
```

```
/*****************************************************************************
 Copy elements of disk1 to disk2.
*****************************************************************************/
void CopyDisk(DISK disk1, DISK *disk2)
{
 disk2 -> radius = disk1.radius;
 disk2 -> center.xVal = disk1.center.xVal;
 disk2 -> center.yVal = disk1.center.yVal;
 disk2 -> center.zVal = disk1.center.zVal;
 disk2 -> density = disk1.density;
 disk2 -> temperature = disk1.temperature;
}

/*****************************************************************************
 Check each line of data as it is read in from the input data file.  If
 data is good, return the type.  If the data is bad, exit the program
 with an error message.
*****************************************************************************/
int CheckDataLine( char * dataLine )
{
 int checkedOK;
 char revisedLine[ MAX_DATA_LINE_LENGTH + 1 ];

 //Strip any leading white space
 //sscanf( dataLine, "%*[ \t\n]%s", &revisedLine );
 //sscanf( dataLine, "%[^\0]", &revisedLine );
 //cout << "revLine: #" << revisedLine << "#" << endl;

       strcpy( revisedLine, dataLine );
 //If the line had nothing but whitespace, get the next line.
 if( strlen( revisedLine ) == 0 )
 {
  checkedOK = BLANK_LINE;
 }
 else if( revisedLine[0] == 'H' )
 {
  checkedOK = HEADER_LINE;
 }
 else if( revisedLine[0] == 'T' )
 {
  checkedOK = TITLE;
 }
 else if( revisedLine[0] == 'E' )
 {
  checkedOK = END_PLASMA_LOOP;
 }
 else if( GoodDataLine( revisedLine ) )
 {
  checkedOK = GOOD_DATA;
 }
 else
 {
```

```
    cerr << "\nIncorrect line in input file: " << endl;
    cerr << dataLine << endl;
    exit( INVALID_DATA );
  }

  return( checkedOK );
}

/*****************************************************************************
  Determine that all six data values are within the required ranges.  And,
  that there are six values.
*****************************************************************************/
int GoodDataLine( char * revisedLine )
{
  int checkedOK = FALSE, num;
  double rad, x, y, z, den, temp;

  num = sscanf( revisedLine, "%lf%lf%lf%lf%lf%lf", &rad, &x, &y, &z, &den,
                                  &temp );

  if( ( 0.0 < rad ) &&  ( 0.0 < den ) &&
     ( 0.0 < temp ) && ( temp < 15e+6 ) && (num == 6 ) )
  {
   checkedOK = TRUE;
  }

  return( checkedOK );
}

/*****************************************************************************
  Make a plasma loop from individual tapered, truncated cylinders.
*****************************************************************************/
int MakePlasmaLoop( ifstream inFile )
{
  DISK testDisk0, testDisk1, testDisk2, testDisk3;
  char dataLine[ MAX_DATA_LINE_LENGTH + 1 ];
  int dataCheck;
  int dum = 0;

  inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );

  //Initially search for the first data line.
        //It takes four lines of data to make one cylinder.
  //So read in three lines to start things off.
  while( ( CheckDataLine( dataLine ) != GOOD_DATA ) && !inFile.eof() )
  {
   inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
  }
  sscanf( dataLine, "%lf%lf%lf%lf%lf%lf", &testDisk1.radius,
      &testDisk1.center.xVal, &testDisk1.center.yVal, &testDisk1.center.zVal,
      &testDisk1.density, &testDisk1.temperature );
```

```
 inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
 while ( ( CheckDataLine( dataLine ) != GOOD_DATA  ) && !inFile.eof() )
 {
  inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
 }
 sscanf( dataLine, "%lf%lf%lf%lf%lf%lf", &testDisk2.radius,
     &testDisk2.center.xVal, &testDisk2.center.yVal, &testDisk2.center.zVal,
     &testDisk2.density, &testDisk2.temperature );

 inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
 while ( ( CheckDataLine( dataLine ) != GOOD_DATA  ) && !inFile.eof() )
 {
  inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
 }
 sscanf( dataLine, "%lf%lf%lf%lf%lf%lf", &testDisk3.radius,
     &testDisk3.center.xVal, &testDisk3.center.yVal, &testDisk3.center.zVal,
     &testDisk3.density, &testDisk3.temperature );

 while ( !inFile.eof() && ( dataCheck != END_PLASMA_LOOP ) )
 {
  CopyDisk( testDisk1, &testDisk0 );
  CopyDisk( testDisk2, &testDisk1 );
  CopyDisk( testDisk3, &testDisk2 );
  inFile.getline( dataLine, MAX_DATA_LINE_LENGTH );
  if( ( dataCheck = CheckDataLine( dataLine ) ) == GOOD_DATA )
  {
   sscanf( dataLine, "%lf%lf%lf%lf%lf%lf", &testDisk3.radius,
    &testDisk3.center.xVal, &testDisk3.center.yVal, &testDisk3.center.zVal,
    &testDisk3.density, &testDisk3.temperature );
   // Apply sensitivity function to temperatures of testDisk1 and testDisk2.
   // This will give the value that will actually be used in deciding the
   // color of the pixels from this sylinder.
   ApplySensitivityFcn( sensFcn, &testDisk1.temperature );
   ApplySensitivityFcn( sensFcn, &testDisk2.temperature );
   MakeCylinder( testDisk0, testDisk1, testDisk2, testDisk3 );
  }
 } //end of loop to make all cylinders in one plasma loop.
 if( inFile.eof() )
 {
  dataCheck = END_OF_FILE;
 }
 return( dataCheck );
}

/*****************************************************************************
 Create an image file in bmp format.
*****************************************************************************/
void SaveImage( GLubyte xrayData[HEIGHT][WIDTH], ofstream imageOut )
{
 char buff[80];
 char color[4];
```

```
struct {
  WORD    bfType;  //file type, must be BM
  DWORD   bfSize;  //size of this file in bytes
  WORD    bfReserved1;  //reserved and must = 0
  WORD    bfReserved2;  //reserved and must = 0
  DWORD   bfOffBytes;  //byte offset from start to the actual bitmap
} bitMapFileHeader;

bitMapFileHeader.bfType = 0x4D42;  //BM
//[bytes]bitMapFileHeader + bmiHeader + colortable + image
bitMapFileHeader.bfSize =  14 + 40 + ( 256 * 4 ) + ( WIDTH * HEIGHT );
bitMapFileHeader.bfReserved1 = 0;
bitMapFileHeader.bfReserved2 = 0;
//[BYTES]bmiHeader + colortable
bitMapFileHeader.bfOffBytes = 14 + ( 10 * 4 ) + ( 256 * 4 );
struct {
  DWORD biSize;  //# of bytes required by this structure
  DWORD biWidth;  //width of bitmap in pixels
  DWORD biHeight;  //height of bitmap in pixels
  WORD biPlanes;  //# of planes for target, must = 1
  WORD biBitCount;  //# of bits per pixel
  DWORD biCompression;  //Type of compression = 0 for no compression
  DWORD biSizeImage;  //# of bytes in the image
  DWORD biXPelsPerMeter;  //Horizontal resolution in pixels per meter
  DWORD biYPelsPerMeter;  //Vertical resolution in pixels per meter
  DWORD biClrUsed;  //color table reference.  = 0 for immediate table
  DWORD biClrImportant;  //set to 0 if all colors are important
} bmiHeader;
bmiHeader.biSize = 40;
bmiHeader.biWidth = WIDTH;
bmiHeader.biHeight = HEIGHT;
bmiHeader.biPlanes = 1;
bmiHeader.biBitCount = 8;
bmiHeader.biCompression = 0;
bmiHeader.biSizeImage = WIDTH * HEIGHT;
bmiHeader.biXPelsPerMeter = 3000;
bmiHeader.biYPelsPerMeter = 3000;
bmiHeader.biClrUsed = 0;
bmiHeader.biClrImportant = 0;

imageOut.setf(ios::binary);
//Put the bitMapFileHeader in the bmp file.
memcpy(buff, &bitMapFileHeader.bfType, 2);
memcpy(buff+2, &bitMapFileHeader.bfSize, 4);
memcpy(buff+6, &bitMapFileHeader.bfReserved1, 2);
memcpy(buff+8, &bitMapFileHeader.bfReserved2, 2);
memcpy(buff+10, &bitMapFileHeader.bfOffBytes, 4);
imageOut.write(buff, 14);
//Put the bmiHeader in the bmp file.
memcpy(buff, &bmiHeader.biSize, 4);
memcpy(buff+4, &bmiHeader.biWidth, 4);
memcpy(buff+8, &bmiHeader.biHeight, 4);
```

```
 memcpy(buff+12, &bmiHeader.biPlanes, 2);
 memcpy(buff+14, &bmiHeader.biBitCount, 2);
 memcpy(buff+16, &bmiHeader.biCompression, 4);
 memcpy(buff+20, &bmiHeader.biSizeImage, 4);
 memcpy(buff+24, &bmiHeader.biXPelsPerMeter, 4);
 memcpy(buff+24, &bmiHeader.biYPelsPerMeter, 4);
 memcpy(buff+28, &bmiHeader.biClrUsed, 4);
 memcpy(buff+32, &bmiHeader.biClrImportant, 4);
 imageOut.write(buff, 40);
 //Put the colormap in the bmp file.
 color[3] = 0x00; //The fourth byte of the quad is zero.
 for( int i = 0; i < 256; i++ )
 {
  color[0] = (unsigned char)(bluemap[i] * 255.0);
  color[1] = (unsigned char)(greenmap[i] * 255.0);
  color[2] = (unsigned char)(redmap[i] * 255.0);
  imageOut.write(color, 4);
 }

 //Put the pixels in the bmp file.  Starting with the lower left pixel and working
 //up one row at a time.  The length of a row must be a multiple of 4 bytes.
 for( i = 0; i < HEIGHT; i++ )
 {
  for( int j = 0; j < WIDTH; j++ )
  {
   imageOut.write( &xrayData[i][j], 1 );
  }
 }

 imageOut.close();
}

/*****************************************************************************
 Map the temperature value into a sensitivity value using the sensitivity
 function specified.  New functions will be added to this switch statement
 as they are needed.
*****************************************************************************/
void ApplySensitivityFcn( int sensFcn, GLdouble * temp )
{
 switch( sensFcn )
 {
  case 0:
   if( *temp < 1.0e+6 )
   {
    *temp = 0.0;
   }
   else
   {
    *temp = 1.0;
   }
   break;
  case 1:
```

```
      *temp = yohkoh_instr_rsp_00( *temp );
      break;
     case 2:
      *temp = yohkoh_instr_rsp_01( *temp );
      break;
     case 3:
      *temp = yohkoh_instr_rsp_02( *temp );
      break;
     case 4:
      *temp = yohkoh_instr_rsp_03( *temp );
      break;
     case 5:
      *temp = yohkoh_instr_rsp_04( *temp );
      break;
     case 6:
      *temp = yohkoh_instr_rsp_05( *temp );
      break;
     default:
      if( *temp < 1.0e+6 )
      {
       *temp = 0.0;
      }
      else
      {
       *temp = 1.0;
      }
   }
}

 /* Routines for T-response of Yohkoh
  given T in degrees K returns f(T)
  in 10^{-28} DN/( sec YkP BCD )
    YkP = Yohkoh pixel
    BCD = brems. col. depth = int(n_e^2)dl in cm^{-5}
  routines include:
     yohkoh_instr_rsp_00 : open
     yohkoh_instr_rsp_01 : Al.1
     yohkoh_instr_rsp_02 : Al/Mg
     yohkoh_instr_rsp_03 : Mg3
     yohkoh_instr_rsp_04 : Al12
     yohkoh_instr_rsp_05 : Be119 */

static double yohkoh_instr_rsp_00( double temp )
/* response code for Yohkoh open */
{
 int i;
 double lgt, frc, f;
 static double dlog10t =    0.0500002;
 static double tv[51] = {
  5.50000e+00,  5.55000e+00,  5.60000e+00,  5.65000e+00,  5.70000e+00,
  5.75000e+00,  5.80000e+00,  5.85000e+00,  5.90000e+00,  5.95000e+00,
  6.00000e+00,  6.05000e+00,  6.10000e+00,  6.15000e+00,  6.20000e+00,
```

```
 6.25000e+00, 6.30000e+00, 6.35000e+00, 6.40000e+00, 6.45000e+00,
 6.50000e+00, 6.55000e+00, 6.60000e+00, 6.65000e+00, 6.70000e+00,
 6.75000e+00, 6.80000e+00, 6.85000e+00, 6.90000e+00, 6.95000e+00,
 7.00000e+00, 7.05000e+00, 7.10000e+00, 7.15000e+00, 7.20000e+00,
 7.25000e+00, 7.30000e+00, 7.35000e+00, 7.40000e+00, 7.45000e+00,
 7.50000e+00, 7.55000e+00, 7.60000e+00, 7.65000e+00, 7.70000e+00,
 7.75000e+00, 7.80000e+00, 7.85000e+00, 7.90000e+00, 7.95000e+00,
 8.00000e+00};
static double fv[51] = {
 1.00895e-02, 2.40323e-02, 5.79735e-02, 1.35350e-01, 2.93222e-01,
 6.12964e-01, 1.24098e+00, 2.35314e+00, 4.12038e+00, 6.77274e+00,
 1.03732e+01, 1.44994e+01, 1.89599e+01, 2.37722e+01, 2.98577e+01,
 4.26198e+01, 7.66350e+01, 1.27973e+02, 1.79364e+02, 2.35745e+02,
 2.90719e+02, 3.45278e+02, 3.92810e+02, 4.25096e+02, 4.37849e+02,
 4.24666e+02, 3.96204e+02, 3.61350e+02, 3.30211e+02, 3.03990e+02,
 2.81867e+02, 2.58477e+02, 2.37962e+02, 2.19285e+02, 2.03900e+02,
 1.92195e+02, 1.83391e+02, 1.76720e+02, 1.71596e+02, 1.67501e+02,
 1.64061e+02, 1.60991e+02, 1.58136e+02, 1.55410e+02, 1.52718e+02,
 1.50017e+02, 1.47336e+02, 1.44690e+02, 1.42077e+02, 1.39497e+02,
 1.36963e+02};

 lgt = log10(temp);
 i = (int)floor( ( lgt - tv[0] )/dlog10t );
 if( i < 0 ) return(0.0);
 if( i > 49 ) return( fv[50] );
 frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
 f = frc*fv[i+1] + (1.0-frc)*fv[i];
 return(f);
}

static double yohkoh_instr_rsp_01( double temp )
/* response code for Yohkoh Al.1 */
{
 int i;
 double lgt, frc, f;
 static double dlog10t =    0.0500002;
 static double tv[51] = {
  5.50000e+00, 5.55000e+00, 5.60000e+00, 5.65000e+00, 5.70000e+00,
  5.75000e+00, 5.80000e+00, 5.85000e+00, 5.90000e+00, 5.95000e+00,
  6.00000e+00, 6.05000e+00, 6.10000e+00, 6.15000e+00, 6.20000e+00,
  6.25000e+00, 6.30000e+00, 6.35000e+00, 6.40000e+00, 6.45000e+00,
  6.50000e+00, 6.55000e+00, 6.60000e+00, 6.65000e+00, 6.70000e+00,
  6.75000e+00, 6.80000e+00, 6.85000e+00, 6.90000e+00, 6.95000e+00,
  7.00000e+00, 7.05000e+00, 7.10000e+00, 7.15000e+00, 7.20000e+00,
  7.25000e+00, 7.30000e+00, 7.35000e+00, 7.40000e+00, 7.45000e+00,
  7.50000e+00, 7.55000e+00, 7.60000e+00, 7.65000e+00, 7.70000e+00,
  7.75000e+00, 7.80000e+00, 7.85000e+00, 7.90000e+00, 7.95000e+00,
  8.00000e+00};
 static double fv[51] = {
  2.77461e-03, 6.97743e-03, 1.79822e-02, 4.48026e-02, 1.03533e-01,
  2.30290e-01, 4.94470e-01, 9.90444e-01, 1.81805e+00, 3.11924e+00,
  4.96336e+00, 7.20120e+00, 9.77824e+00, 1.27677e+01, 1.68619e+01,
```

```
  2.63553e+01,  5.24393e+01,  9.21794e+01,  1.32344e+02,  1.76815e+02,
  2.20381e+02,  2.63922e+02,  3.02167e+02,  3.28608e+02,  3.39777e+02,
  3.30641e+02,  3.09458e+02,  2.83106e+02,  2.59439e+02,  2.39324e+02,
  2.22148e+02,  2.03692e+02,  1.87373e+02,  1.72428e+02,  1.60076e+02,
  1.50666e+02,  1.43582e+02,  1.38216e+02,  1.34095e+02,  1.30807e+02,
  1.28051e+02,  1.25597e+02,  1.23322e+02,  1.21157e+02,  1.19026e+02,
  1.16894e+02,  1.14783e+02,  1.12705e+02,  1.10659e+02,  1.08644e+02,
  1.06669e+02};

 lgt = log10(temp);
 i = (int)floor( ( lgt - tv[0] )/dlog10t );
 if( i < 0 ) return(0.0);
 if( i > 49 ) return( fv[50] );
 frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
 f = frc*fv[i+1] + (1.0-frc)*fv[i];
 return(f);
}

static double yohkoh_instr_rsp_02( double temp )
/* response code for Yohkoh Al/Mg */
{
 int i;
 double lgt, frc, f;
 static double dlog10t =    0.0500002;
 static double tv[51] = {
  5.50000e+00,  5.55000e+00,  5.60000e+00,  5.65000e+00,  5.70000e+00,
  5.75000e+00,  5.80000e+00,  5.85000e+00,  5.90000e+00,  5.95000e+00,
  6.00000e+00,  6.05000e+00,  6.10000e+00,  6.15000e+00,  6.20000e+00,
  6.25000e+00,  6.30000e+00,  6.35000e+00,  6.40000e+00,  6.45000e+00,
  6.50000e+00,  6.55000e+00,  6.60000e+00,  6.65000e+00,  6.70000e+00,
  6.75000e+00,  6.80000e+00,  6.85000e+00,  6.90000e+00,  6.95000e+00,
  7.00000e+00,  7.05000e+00,  7.10000e+00,  7.15000e+00,  7.20000e+00,
  7.25000e+00,  7.30000e+00,  7.35000e+00,  7.40000e+00,  7.45000e+00,
  7.50000e+00,  7.55000e+00,  7.60000e+00,  7.65000e+00,  7.70000e+00,
  7.75000e+00,  7.80000e+00,  7.85000e+00,  7.90000e+00,  7.95000e+00,
  8.00000e+00};
 static double fv[51] = {
  2.07817e-04,  6.80402e-04,  2.05662e-03,  5.86829e-03,  1.55755e-02,
  3.94193e-02,  9.45655e-02,  2.09322e-01,  4.18741e-01,  7.75536e-01,
  1.32438e+00,  2.06871e+00,  3.04000e+00,  4.31498e+00,  6.21259e+00,
  1.08623e+01,  2.38189e+01,  4.39982e+01,  6.52788e+01,  8.98284e+01,
  1.14151e+02,  1.39553e+02,  1.62925e+02,  1.80595e+02,  1.90506e+02,
  1.89800e+02,  1.82796e+02,  1.72811e+02,  1.63677e+02,  1.55329e+02,
  1.47332e+02,  1.37159e+02,  1.27405e+02,  1.17853e+02,  1.09646e+02,
  1.03261e+02,  9.84019e+01,  9.47082e+01,  9.18763e+01,  8.96294e+01,
  8.77615e+01,  8.61131e+01,  8.45952e+01,  8.31563e+01,  8.17426e+01,
  8.03298e+01,  7.89331e+01,  7.75595e+01,  7.62076e+01,  7.48760e+01,
  7.35709e+01};

 lgt = log10(temp);
 i = (int)floor( ( lgt - tv[0] )/dlog10t );
 if( i < 0 ) return(0.0);
```

```
 if( i > 49 ) return( fv[50] );
 frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
 f = frc*fv[i+1] + (1.0-frc)*fv[i];
 return(f);
}

static double yohkoh_instr_rsp_03( double temp )
/* response code for Yohkoh Mg3 */
{
 int i;
 double lgt, frc, f;
 static double dlog10t = 0.0500002;
 static double tv[51] = {
  5.50000e+00, 5.55000e+00, 5.60000e+00, 5.65000e+00, 5.70000e+00,
  5.75000e+00, 5.80000e+00, 5.85000e+00, 5.90000e+00, 5.95000e+00,
  6.00000e+00, 6.05000e+00, 6.10000e+00, 6.15000e+00, 6.20000e+00,
  6.25000e+00, 6.30000e+00, 6.35000e+00, 6.40000e+00, 6.45000e+00,
  6.50000e+00, 6.55000e+00, 6.60000e+00, 6.65000e+00, 6.70000e+00,
  6.75000e+00, 6.80000e+00, 6.85000e+00, 6.90000e+00, 6.95000e+00,
  7.00000e+00, 7.05000e+00, 7.10000e+00, 7.15000e+00, 7.20000e+00,
  7.25000e+00, 7.30000e+00, 7.35000e+00, 7.40000e+00, 7.45000e+00,
  7.50000e+00, 7.55000e+00, 7.60000e+00, 7.65000e+00, 7.70000e+00,
  7.75000e+00, 7.80000e+00, 7.85000e+00, 7.90000e+00, 7.95000e+00,
  8.00000e+00};
 static double fv[51] = {
  5.41356e-06, 2.20848e-05, 8.12592e-05, 2.78793e-04, 8.90374e-04,
  2.62938e-03, 7.21077e-03, 1.81674e-02, 4.22954e-02, 9.29136e-02,
  1.93202e-01, 3.78956e-01, 7.05772e-01, 1.27351e+00, 2.36600e+00,
  5.83541e+00, 1.65382e+01, 3.36555e+01, 5.19335e+01, 7.32568e+01,
  9.46421e+01, 1.16887e+02, 1.37194e+02, 1.52145e+02, 1.59679e+02,
  1.57264e+02, 1.48831e+02, 1.37717e+02, 1.27556e+02, 1.18359e+02,
  1.09693e+02, 9.94492e+01, 8.99840e+01, 8.11926e+01, 7.39861e+01,
  6.86123e+01, 6.46433e+01, 6.16809e+01, 5.94385e+01, 5.76801e+01,
  5.62177e+01, 5.49208e+01, 5.37420e+01, 5.26525e+01, 5.16035e+01,
  5.05703e+01, 4.95682e+01, 4.86035e+01, 4.76696e+01, 4.67631e+01,
  4.58902e+01};

 lgt = log10(temp);
 i = (int)floor( ( lgt - tv[0] )/dlog10t );
 if( i < 0 ) return(0.0);
 if( i > 49 ) return( fv[50] );
 frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
 f = frc*fv[i+1] + (1.0-frc)*fv[i];
 return(f);
}

static double yohkoh_instr_rsp_04( double temp )
/* response code for Yohkoh Al12 */
{
 int i;
 double lgt, frc, f;
 static double dlog10t = 0.0500002;
```

```
  static double tv[51] = {
   5.50000e+00,  5.55000e+00,  5.60000e+00,  5.65000e+00,  5.70000e+00,
   5.75000e+00,  5.80000e+00,  5.85000e+00,  5.90000e+00,  5.95000e+00,
   6.00000e+00,  6.05000e+00,  6.10000e+00,  6.15000e+00,  6.20000e+00,
   6.25000e+00,  6.30000e+00,  6.35000e+00,  6.40000e+00,  6.45000e+00,
   6.50000e+00,  6.55000e+00,  6.60000e+00,  6.65000e+00,  6.70000e+00,
   6.75000e+00,  6.80000e+00,  6.85000e+00,  6.90000e+00,  6.95000e+00,
   7.00000e+00,  7.05000e+00,  7.10000e+00,  7.15000e+00,  7.20000e+00,
   7.25000e+00,  7.30000e+00,  7.35000e+00,  7.40000e+00,  7.45000e+00,
   7.50000e+00,  7.55000e+00,  7.60000e+00,  7.65000e+00,  7.70000e+00,
   7.75000e+00,  7.80000e+00,  7.85000e+00,  7.90000e+00,  7.95000e+00,
   8.00000e+00};
  static double fv[51] = {
   1.28114e-12,  1.38595e-11,  4.80228e-10,  4.41040e-09,  4.73539e-08,
   3.58486e-07,  2.96677e-06,  1.56400e-05,  7.17745e-05,  2.93496e-04,
   1.09007e-03,  3.56551e-03,  1.01255e-02,  2.59010e-02,  6.25827e-02,
   1.62521e-01,  4.43389e-01,  9.62890e-01,  1.68332e+00,  2.67787e+00,
   3.61954e+00,  4.81358e+00,  6.08095e+00,  7.33778e+00,  8.59615e+00,
   9.85502e+00,  1.12021e+01,  1.24850e+01,  1.35449e+01,  1.41905e+01,
   1.45598e+01,  1.45653e+01,  1.43720e+01,  1.38685e+01,  1.32029e+01,
   1.25454e+01,  1.19659e+01,  1.14821e+01,  1.10789e+01,  1.07338e+01,
   1.04341e+01,  1.01662e+01,  9.91788e+00,  9.68007e+00,  9.45109e+00,
   9.22976e+00,  9.01683e+00,  8.81230e+00,  8.61912e+00,  8.43765e+00,
   8.26690e+00};

  lgt = log10(temp);
  i = (int)floor( ( lgt - tv[0] )/dlog10t );
  if( i < 0 ) return(0.0);
  if( i > 49 ) return( fv[50] );
  frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
  f = frc*fv[i+1] + (1.0-frc)*fv[i];
  return(f);
}


static double yohkoh_instr_rsp_05( double temp )
/* response code for Yohkoh Be119 */
{
  int i;
  double lgt, frc, f;
  static double dlog10t = 0.0500002;
  static double tv[51] = {
   5.50000e+00,  5.55000e+00,  5.60000e+00,  5.65000e+00,  5.70000e+00,
   5.75000e+00,  5.80000e+00,  5.85000e+00,  5.90000e+00,  5.95000e+00,
   6.00000e+00,  6.05000e+00,  6.10000e+00,  6.15000e+00,  6.20000e+00,
   6.25000e+00,  6.30000e+00,  6.35000e+00,  6.40000e+00,  6.45000e+00,
   6.50000e+00,  6.55000e+00,  6.60000e+00,  6.65000e+00,  6.70000e+00,
   6.75000e+00,  6.80000e+00,  6.85000e+00,  6.90000e+00,  6.95000e+00,
   7.00000e+00,  7.05000e+00,  7.10000e+00,  7.15000e+00,  7.20000e+00,
   7.25000e+00,  7.30000e+00,  7.35000e+00,  7.40000e+00,  7.45000e+00,
   7.50000e+00,  7.55000e+00,  7.60000e+00,  7.65000e+00,  7.70000e+00,
   7.75000e+00,  7.80000e+00,  7.85000e+00,  7.90000e+00,  7.95000e+00,
```

```
 8.00000e+00};
 static double fv[51] = {
  1.62282e-18,  8.69929e-17,  3.51845e-15,  1.03648e-13,  2.41363e-12,
  4.21862e-11,  1.63456e-09,  1.54806e-08,  1.48222e-07,  1.06793e-06,
  7.02349e-06,  3.46967e-05,  1.41041e-04,  4.99637e-04,  1.62981e-03,
  4.89867e-03,  1.32784e-02,  3.15088e-02,  6.68034e-02,  1.30267e-01,
  2.05412e-01,  3.34308e-01,  5.20663e-01,  7.78464e-01,  1.12565e+00,
  1.57786e+00,  2.14043e+00,  2.79409e+00,  3.52412e+00,  4.34205e+00,
  5.15012e+00,  5.90210e+00,  6.55741e+00,  7.07878e+00,  7.49269e+00,
  7.82844e+00,  8.12285e+00,  8.39323e+00,  8.65198e+00,  8.90404e+00,
  9.15081e+00,  9.38752e+00,  9.60492e+00,  9.79802e+00,  9.96492e+00,
  1.01053e+01,  1.02189e+01,  1.03053e+01,  1.03637e+01,  1.03947e+01,
  1.03993e+01};

 lgt = log10(temp);
 i = (int)floor( ( lgt - tv[0] )/dlog10t );
 if( i < 0 ) return(0.0);
 if( i > 49 ) return( fv[50] );
 frc = ( lgt - tv[i] )/( tv[i+1] - tv[i] );
 f = frc*fv[i+1] + (1.0-frc)*fv[i];
 return(f);
}


//The functions below are not used, but may be of some value in later attempts
//to manipulate the pixel buffers.
/*int createData( GLubyte xrayData[HEIGHT][WIDTH] )
{
        srand( 97 ); //Fixed seed value results in same set every time.
        for( int i = 0; i < HEIGHT; i++ )
        {
          for( int j = 0; j < WIDTH; j++ )
          {
                  xrayData[i][j] = (GLubyte)(rand()%255);
                  //xrayData[i][j] = (GLubyte)0;
    //xrayData[i][j] = (GLubyte)j; //This shows the total range of colors
          }
   //commented out below
   for( j = 100; j < 150; j++ )
          {
                  xrayData[i][j] = (GLubyte)150;
          }
          for( j = 150; j < 200; j++ )
          {
                  xrayData[i][j] = (GLubyte)200;
          }
          for( j = 200; j < WIDTH; j++ )
          {
                  xrayData[i][j] = (GLubyte)255;
          }
   //end of commented out
        }
```

```
 return (1);
}

void playData( GLubyte xrayData[HEIGHT][WIDTH] )
{
 int num;

 for( int i = 1; i < 99; i+=3 )
 {
  for( int j = 1; j < WIDTH-1; j+=3 )
  {
   if( (num = checkLocal9( xrayData, i, j )) > 4 )
    increaseLocal9( xrayData, i, j, 50 );
   else
    if( num <= 3 )
     increaseLocal9( xrayData, i, j, -50 );
  }
 }
}

int checkLocal9( GLubyte xrayData[HEIGHT][WIDTH], int hpos, int wpos )
{
 int count = 0;

 for( int i = hpos-1; i < hpos+2; i++ )
 {
  for( int j = wpos-1; j < wpos+2; j++ )
  {
   if( xrayData[i][j] > (GLubyte)220 ) count++;
  }
 }
 return (count);
}

void increaseLocal9( GLubyte xrayData[HEIGHT][WIDTH], int hpos, int wpos, int amt )
{
 int temp;

 for( int i = hpos-1; i < hpos+2; i++ )
 {
  for( int j = wpos-1; j < wpos+2; j++ )
  {
   temp = (int)xrayData[i][j];
   temp += amt;
   if( temp > 255 )
   {
    temp = 255;
   }
   else if( temp < 0 )
   {
    temp = 0;
   }
```

```
    xrayData[i][j] = (GLubyte)temp;
   }
  }
}


int readData( GLubyte xrayData[HEIGHT][WIDTH], ifstream *infile )
{
        for( int i = 0; i < HEIGHT; i++ )
        {
          for( int j = 0; j < WIDTH; j++ )
          {
                  infile->get( xrayData[j][i] );
          }
        }
        return( infile->fail() );
}*/
```

APPENDIX E

MATRIX.CPP

```
/*=============================================================================

  matrix.cpp     Steve Lundberg     11/12/97

  This file contains utility functions having to do with matrix
  operations.

  Functions included:

    InvertMatrix()   -- uses Gauss-Jordan elimination method
    SwapRows()       -- used by InvertMatrix()
    SingularMatrix() -- used by InvertMatrix()
    Pivot()          -- used by InvertMatrix()
    PrintMatrix()
    MatrixMult()
    TransformPoint()

=============================================================================*/

#include <stdlib.h>
#include <fstream.h>
#include <process.h>
#include <math.h>

#ifdef WIN32
#include <afxwin.h>
#include <wingdi.h>
#endif

#include <GL\gl.h>
#include <GL\glu.h>
#include <GL\glaux.h>

#include "sxi.h"

/*****************************************************************************
  Invert matrix - calc the inverse of the current model matrix so can map
  from screen coordinates to current cylinder coordinates.  First retrieve
  the matrix, then invert it.  The inverted matrix is returned at the
  supplied pointer.

  From Press, W.H., Numerical Recipes in C, Second Edition:

  Gauss-Jordan Elimination.  G-J Elim. is as good as any other technique for
  inverting matrices.  For the closely related case of solving sets of linear
  equations, LU decomposition is better.

  In any matrix inverting operation, one needs to be careful of singular
  matrices.  Initially I will not support singular matrices, but the technique
  of Singular Value Decomposition (SVD) discussed in Press will handle some
  kinds of singular matrices.

  G-J Elim. with pivoting starts with the upper left element of the matrix.
```

An equally-sized identity matrix is also used. Operations on the initial matrix are also performed on this identity matrix - which is then no longer the identity matrix. The goal of this procedure is to transform the initial matrix into an identity matrix by using only linear operations. When the initial matrix has been transformed to the identity matrix, the original identity matrix is the inverse matrix of the initial matrix.

The pivoting is performed to guarantee the mathematical stability of this operation. If pivoting is not used, there is a substantial probability of roundoff and truncation errors dominating the solution obtained.

First pivot the rows of the matrix to place the largest element in column 1 in the top position. Next divide the top row by the first element. This makes the first element = 1. The top row of the identity matrix is also divided by this element. Now all other rows in the matrix are individually added to a multiple of the first row chosen so that the resulting first element will be zero. After completing this operation the entire first column has been reduced to a 1 on the diagonal (the first element in this case) and zeroes for all other elements. This sequence is now repeated for each of the rest of the columns.

The transformation matrices in openGL are stored in column-major order. This means that A[1] is the first element of the second row. A[4] is the second element of the first row.

i.e. A = A[0] A[4] A[8] A[12]
A[1] A[5] A[9] A[13]
A[2] A[6] A[10] A[14]
A[3] A[7] A[11] A[15]

```
**************************************************************************/
void InvertMatrix( GLdouble inputMatrix[DIM*DIM],
                   GLdouble outputMatrix[DIM*DIM] )
{
 int row, col, k;
 int pivot;
 GLdouble mat[DIM*DIM], outMat[DIM*DIM];
 GLdouble factor;

 // Copy inputMatrix to temp storage.
 for ( k = 0; k < DIM*DIM; k++ )
 {
  mat[k] = inputMatrix[k];
 }

 // Initialize outMat to the identity matrix.
 for ( k = 0; k < DIM*DIM; k++ )  // Make all zero.
 {
  outMat[k] = 0;
 }
 for ( row = 0; row < DIM; row++ ) // Set diagonals to 1.
 {
```

```
 outMat[row*DIM + row] = 1;
}

// Begin G-J operations

for ( row = 0; row < DIM; row++ )     // Walk down all rows
{
 pivot = Pivot( mat, row );
 if ( pivot != row )    // Pivot if necessary.
 {
  SwapRows( mat, pivot, row );
  SwapRows( outMat, pivot, row );  // Must do same thing to both matrices.
 }

 // Make diagonal element = 1 by dividing by that element.  Divide each
 // element in this row in the original identity matrix (outMat) also.
 if ( mat[ (row*DIM) + row ] != 0.0 )
 {
  factor = mat[ (row*DIM) + row ];
  for ( col = 0; col < DIM; col++ )
  {
      mat[ (col*DIM)+row ] /= factor;
   outMat[ (col*DIM)+row ] /= factor;
  }
 }
 else
 {
  SingularMatrix( mat );
 }

/*-------------------------------------------------------------------------
 Now zero out all elements on this column except the diagonal one by
 subtracting some appropriate fraction of the current row from each of
 the other rows.  Apply same operation to each of the rest of the
 elements in the row being operated on.  Also apply the same operation
 to each element in the original identity matrix (outMat).

 Calculate the factor to mult. the ith row by to then subtract
 from the kth row.  The factor is just the value of the kth row
 element directly above or below the diagonal element of the ith row
 because the ith row element is already 1.
 NOTE: row*DIM gives the first element in a column and adding k gives the
 proper row below that row.
-------------------------------------------------------------------------*/
 for ( k = 0; k < row; k++ )   // For rows above this one.
 {
  // For all elements in the kth row.  The ones in the input matrix
  // to the left of the diagonal in this row will all be zero, but
  // all the elements in the output matrix need to be hit, so the
  // loop will be over all elements.

  // Don't need to do anything if this element is already zero.
```

```
    if ( mat[ (row*DIM) + k ] != 0.0 )
    {
     factor = mat[ (row*DIM) + k ];
     for ( col = 0; col < DIM; col++ )
     {
        mat[ (col*DIM) + k ] -= factor *    mat[ (col*DIM) + row ];
      outMat[ (col*DIM) + k ] -= factor * outMat[ (col*DIM) + row ];
     }
    }
   }

   for ( k = row+1; k < DIM; k++ ) // For rows below this one.
   {
    // Don't need to do anything if this element is already zero.
    if ( mat[ (row*DIM) + k ] != 0.0 )
    {
     factor = mat[ (row*DIM) + k ];  // Matrix element on col of diag in row.
     for ( col = 0; col < DIM; col++ )
     {
        mat[ (col*DIM) + k ] -= factor *    mat[ (col*DIM) + row ];
      outMat[ (col*DIM) + k ] -= factor * outMat[ (col*DIM) + row ];
     }
    }
   }
  } // End for loop over all rows.  Now outMat[] is the inverse of mat and mat
    // is reduced to the identity matrix.

  // Copy outMat[] to outputMatrix[].
  for ( row = 0; row < DIM*DIM; row++ )
  {
   outputMatrix[row] = outMat[row];
  }
} // End invertMatrix().

/*****************************************************************************
 Swap rows in the matrix to put the row with the largest value on the
 diagonal.  Only consider rows below the current one.  Performed by
 swapping the elements in the current matrix.
*****************************************************************************/
void SwapRows( GLdouble mat[], int pivot, int row )
{
 int col;
 GLdouble temp;

 for ( col = 0; col < DIM; col++ )
 {
  temp = mat[ pivot + (col*DIM) ];
  mat[ pivot + (col*DIM) ] = mat[ row + (col*DIM) ];
  mat[ row + (col*DIM) ] = temp;
 }
}
```

```
/*****************************************************************************
 Exit if the matrix is singular.
*****************************************************************************/
void SingularMatrix( GLdouble mat[] )
{
 cout << endl << "Matrix is singular!  Cannot continue." << endl;
 PrintMatrix( mat );
 exit( SINGULAR_MATRIX );
}

/*****************************************************************************
 Find the largest element below or on the diagonal indexed by row.
 The pivot must be chosen at the time it is needed from those available.
*****************************************************************************/
int Pivot( GLdouble mat[], int row )
{
 int pivot, ckRow;
 GLdouble max;

 if ( row < DIM-1 )  // The last row cannot be swapped.
 {
  max = fabs( mat[ (row*DIM) + row ] );  // Set diagonal element as the max.
  pivot = row;
  for ( ckRow = row+1; ckRow < DIM; ckRow++ )  // Check rows below this row.
  {
   if ( fabs( mat[ (row*DIM) + ckRow ] ) > max )
   {
    max = fabs( mat[ (row*DIM) + ckRow ] );
    pivot = ckRow;
   }
  }
 }
 else
 {
  pivot = DIM-1;  // The last row cannot be swapped with anyone.
 }
 return ( pivot );
}


/*****************************************************************************
 Print the given matrix.
*****************************************************************************/
void PrintMatrix( GLdouble mat[] )
{
 int row, col;

 for ( row = 0; row < DIM; row++ )
 {
  for ( col = 0; col < DIM; col++ )
  {
   printf( "%4.2f  ", mat[ row + (col*DIM) ] );
```

```
    }
    cout << endl;
  }
 }

/*****************************************************************************
 Multiply inMat1 * inMat2 -> outMat
*****************************************************************************/
void MatrixMult( GLdouble inMat1[], GLdouble inMat2[], GLdouble outMat[] )
{
 int row, col, k;

 for ( row = 0; row < DIM; row++ )
 {
  for ( col = 0; col < DIM; col++ )
  {
   outMat[ row + (col*DIM) ] = 0;
   for ( k = 0; k < DIM; k++ )
   {
    outMat[ row + (col*DIM) ] +=
     inMat1[ row + (k*DIM) ] * inMat2[ (col*DIM) + k ];
   }
  }
 }
}


/*****************************************************************************
 Function to mult. matrix by POINT3.  Set w = 1 for the point.
 Definitions of openGL require this to be done as M * Pt, not
 Pt * M.
*****************************************************************************/
POINT3 TransformPoint( POINT3 pt, GLdouble matrix[] )
{
 GLdouble inPt[ DIM ], outPt[ DIM ];
 int row, col;
 POINT3 tranPt;

 // Initialize inPt from pt.
 inPt[ 0 ] = pt.xVal;
 inPt[ 1 ] = pt.yVal;
 inPt[ 2 ] = pt.zVal;
 inPt[ 3 ] = 1.0;

 for ( row = 0; row < DIM; row++ )
 {
  outPt[ row ] = 0.0;
  for ( col = 0; col < DIM; col++ )
  {
   outPt[ row ] += matrix[ row + (col * DIM) ] * inPt[ col ];
  }
 }
```

```
// Copy outPt[] to tranPt and return it.
tranPt.xVal = (GLfloat)outPt[ 0 ];
tranPt.yVal = (GLfloat)outPt[ 1 ];
tranPt.zVal = (GLfloat)outPt[ 2 ];

if ( outPt[3] != 1.0 )
{
 cout << "w not = 1.0 for this Xform!" << endl;
 PrintMatrix( matrix );
}

return ( tranPt );
}
```

APPENDIX F

SXI.H

```
/*==========================================================================
  sxi.h       Steve Lundberg     2/15/97

  Header file for functions used to produce picture from
  x-ray intensity input.
==========================================================================*/

#ifndef sxi_h
#define sxi_h

// Must turn on one of these on the compile command line.
//#define WIN
//#define UNIX

//gluLookAt data.  Also used by cylmap.cpp
extern GLdouble EYEX;
extern GLdouble EYEY;
extern GLdouble EYEZ;

extern GLdouble CENTERX;
extern GLdouble CENTERY;
extern GLdouble CENTERZ;

extern int viewport[];

extern GLfloat xMin;
extern GLfloat xMax;
extern GLfloat yMin;
extern GLfloat yMax;
extern GLfloat zMin;
extern GLfloat zMax;

extern int sensFcn;
extern GLdouble exposure;

// Dimensions of synthetic xray image array in pixels
// NOTE: WIDTH must be a multiple of four bytes or else the
// bmp file will not be written correctly.
// NOTE 2: If the pixel size of the xray image buffer does not match
// exactly the size of the window, then pixel effects WILL be seen.
const int WIDTH = 600;
const int HEIGHT = 600;



const int DIM       = 4;// DIM is the dimension of the square matrix
                    // to be inverted.

const float PI =     (float)3.141593;
const float SQRT2 =   (float)1.4143;

const float RadToDeg = (float)( 360.0 / ( 2.0 * PI ) );
```

```
const int SUCCESS = 1;
const int FAILURE = 0;
//const int TRUE    = 1;  TRUE and FALSE already defined in C++
//const int FALSE   = 0;
const int IMAGINARY_INTERSECTION = 101;
const int LARGE_ANGLE          = 102;
const int SINGULAR_MATRIX      = 103;
const int NO_DATA_FILE         = 104;
const int NO_BMP_FILE          = 105;
const int INVALID_DATA         = 106;
const int NO_INPUT_FILE_NAME   = 107;

const int MAX_DATA_LINE_LENGTH = 80;

const int BLANK_LINE =     0;
const int HEADER_LINE =    1;
const int TITLE =          2;
const int END_PLASMA_LOOP = 3;
const int GOOD_DATA =      4;
const int END_OF_FILE =    5;


extern GLUquadricObj *cylQuadric; //Used for the construction of each cylinder.
extern GLubyte xrayData[HEIGHT][WIDTH]; //Holds the pixel color values.

extern float redmap[256]; //Maps used to map the color values to actual colors.
extern float greenmap[256];
extern float bluemap[256];

extern GLdouble initInverse[ DIM * DIM ]; //Needed by later function calls.

struct POINT3 {
  GLdouble xVal;
  GLdouble yVal;
  GLdouble zVal;
};

struct DISK {
  POINT3 center;
  GLdouble radius;
  GLdouble density;
  GLdouble temperature;
};

struct PLANE {        //Ax + By + Cz + D = 0
  GLdouble xParam;
  GLdouble yParam;
  GLdouble zParam;
  GLdouble offset;
};

struct RAY {
```

```
  POINT3 initPt;      //initPt is starting end of RAY
  POINT3 dirVect;     //dirVect is the direction vector for RAY
};

//in sxi.cpp
void myinit( char *argv[] );
void CALLBACK display( void );
void CALLBACK myReshape( int w, int h );

// in sxiUtils.cpp
int displayData( GLubyte xrayData[HEIGHT][WIDTH] );
void SetupAxes( void );
void CopyDisk( DISK disk1, DISK *disk2 );
int CheckDataLine( char * dataLine );
int GoodDataLine( char * revisedLine );
int MakePlasmaLoop( ifstream inFile );
void SaveImage( GLubyte xrayData[HEIGHT][WIDTH], ofstream imageOut );
void ApplySensitivityFcn( int fcn, GLdouble * temp );

// in cylinder.cpp
void MakeCylinder( DISK prior, DISK bot, DISK top, DISK after );
GLfloat DistBetweenPoints( POINT3 pt1, POINT3 pt2 );
void MoveToOrigin( POINT3 pt );
void AlignZAxis( POINT3 botPt, POINT3 topPt );
void CutPlane( POINT3 left, POINT3 middle, POINT3 right, PLANE * pl );
void ExtendCyl( GLfloat angZ, DISK thisEnd, GLfloat * deltaZ );
void CheckAngle( GLfloat angZ, POINT3 pt );
GLfloat DotProd( PLANE pl1, PLANE pl2 );
void SetEqn( PLANE pl, GLdouble eqn[], GLint up );
GLfloat DistBetweenPoints( POINT3 pt1, POINT3 pt2 );

//in matrix.cpp
void InvertMatrix( GLdouble inputMatrix[16], GLdouble outputMatrix[] );
void SwapRows( GLdouble mat[], int pivot, int i );
void SingularMatrix( GLdouble mat[] );
int Pivot( GLdouble mat[], int row );
void PrintMatrix( GLdouble mat[] );
void MatrixMult( GLdouble inMat1[], GLdouble inMat2[], GLdouble outMat[] );
POINT3 TransformPoint( POINT3 cylPt, GLdouble Matrix[] );

// in cylmap.cpp
void MapCylinderPixels( GLdouble botRad, GLdouble topRad, GLdouble height,
                PLANE cutPl1, PLANE cutPl2, GLdouble density,
                GLdouble sensitivity1, GLdouble sensitivity2 );
void MinMax( GLdouble minMax[], POINT3 scrPt, int *init );
GLdouble CalculateIntersectionDistance( int xPos, int yPos,
                  GLdouble projMat[],
                  GLdouble modelMat[],
                  GLdouble modelMatFixed[],
                  GLdouble botRad, GLdouble topRad, GLdouble height,
                  PLANE cutPl1, PLANE cutPl2 );
int GetEndPoints( POINT3 * posRootPtr, POINT3 * negRootPtr, RAY cylRay,
```

```
            GLdouble * quadricData );
int MovePointsWithinCutPlanes( POINT3 * posRootPtr, POINT3 * negRootPtr,
                    PLANE cutPl1, PLANE cutPl2 );
void IntersectRayWithCutPlane( POINT3 * rootPtr, PLANE cutPl1, RAY cylRay );
void AdjustPixelColor( int xPos, int yPos, int dist );
GLdouble CalcTempFcn( GLdouble temp, int tempFcn );

#endif
```

APPENDIX G

EXAMPLE INITIALIZATION AND DATA FILES

File sxi.ini:

```
0 0 600 600
-0.0 -0.20 0.20 0 0 0 0 0 1
1  .08 FIXED_PIXEL_SIZE 0.14
-15 0 0
```

File format:
line 1: windowX, windowY, windowWidth, windowHeight
line 2: EYEX, EYEY, EYEZ, CENTERX, CENTERY, CENTERZ, UPX, UPY, UPZ
line 3: Sensitivity function index, exposure value, pixel sizing, size
line 4: xOffset, yOffset, zOffset

Line 1 controls the size (in pixels) of the window on the screen. This needs to be the same size as the final synthetic xray image will be or pixel artifact effects will be seen.

Line 2 controls the viewpoint from which the scene is drawn. The three EYE coordinates and the three CENTER coordinates determine the viewing angle. The UP coordinates determine which direction is shown as up on the screen.

Line 3 sets the sensitivity function to be used. This relates to the type of filter in use. And sets the exposure which relates to the brightness of the image. And defines the type of scaling to be used: FIXED_AREA_SIZE will show only that part of the image that is within the fixed boundaries; whereas FIXED_PIXEL_SIZE will show the image as it would look when each pixel covered the stated amount of area on the Sun's surface.

If line 3 has FIXED_AREA_SIZE, then size contains the horizontal span for the fixed area size - the y and z values match the x ones and thus all viewing is of a cube. If line 3 has FIXED_PIXEL_SIZE then the value is the span of one pixel.

Line 4 is used to offset the center of the viewing volume. Place the coordinates of the desired center for the viewing volume here. Be sure to use the same units as for all other dimensions.

NOTE: All length parameters - radius, x, y, z, pixel size, and offset must be in the same units. Positive Z is away from the Sun's surface.

File input_data:

```
4.58318   5.03010   -2.39816   -1.39698   32.5637    318820.
4.58318   5.75791   -2.96428    0.00000   32.5637    318820.
4.35026   5.03010   -2.39816    1.39698   32.5637    928480.
4.23094   4.31004   -1.89874    2.35220   13.6805  1.43248e+06
4.12981   3.58737   -1.36782    3.28825   8.03442  1.86923e+06
4.04729   2.84821   -0.804874   4.19207   5.53772  2.25151e+06
3.98117   2.08341   -0.211612   5.05411   4.18923  2.58864e+06
3.92774   1.29056    0.407562   5.87134   3.36101  2.89004e+06
```

3.88476	0.466850	1.05147	6.63734	2.81452	3.15818e+06
3.85092	-0.393197	1.72146	7.33823	2.44063	3.39148e+06
3.82305	-1.28392	2.41166	7.97835	2.17009	3.59667e+06
3.79963	-2.20407	3.12009	8.55375	1.96982	3.77508e+06
3.77955	-3.15515	3.84647	9.05109	1.82274	3.92444e+06
3.76126	-4.13185	4.58671	9.47237	1.71365	4.04742e+06
3.74373	-5.12978	5.33757	9.81835	1.63281	4.14640e+06
3.72613	-6.14648	6.09683	10.0815	1.57670	4.21953e+06
3.70771	-7.17818	6.86155	10.2552	1.54514	4.26242e+06
3.68810	-8.21885	7.62815	10.3478	1.52860	4.28542e+06
3.67124	-9.05608	8.24168	10.3627	1.52583	4.28929e+06
3.66537	-9.33299	8.44374	10.3495	1.53012	4.28328e+06
3.64236	-10.3782	9.20469	10.2617	1.54684	4.26006e+06
3.61783	-11.4185	9.95867	10.0923	1.58037	4.21463e+06
3.59181	-12.4482	10.7018	9.83369	1.63929	4.13820e+06
3.56476	-13.4626	11.4317	9.49079	1.72221	4.03735e+06
3.53735	-14.4589	12.1467	9.07174	1.83485	3.91146e+06
3.51022	-15.4323	12.8441	8.57609	1.98573	3.75992e+06
3.48440	-16.3766	13.5197	8.00040	2.19180	3.57881e+06
3.46086	-17.2924	14.1744	7.35855	2.47059	3.37085e+06
3.44073	-18.1778	14.8066	6.65424	2.85670	3.13478e+06
3.42629	-19.0261	15.4109	5.88285	3.42333	2.86362e+06
3.41876	-19.8410	15.9889	5.05716	4.28742	2.55883e+06
3.41983	-20.6239	16.5397	4.18340	5.70714	2.21784e+06
3.43276	-21.3748	17.0589	3.26329	8.36948	1.83143e+06
3.46032	-22.0990	17.5435	2.30373	14.5520	1.38892e+06
3.50339	-22.8076	17.9931	1.31576	36.5484	876406.
3.61989	-23.5304	18.4958	0.00000	36.5484	248285.
3.61989	-22.8076	17.9931	-1.31576	36.5484	248285.
END LOOP					
4.80654	4.42275	-6.90290	-2.08676	51.2853	429871.
4.80654	5.48266	-6.74896	0.00000	51.2853	429871.
4.49161	4.42275	-6.90290	2.08676	51.2853	1.23878e+06
4.36181	3.37460	-6.94396	3.53241	21.7752	1.90111e+06
4.26675	2.29320	-6.95393	4.95372	12.8725	2.47262e+06
4.20106	1.15670	-6.92940	6.33112	8.88390	2.97637e+06
4.15867	-0.0496263	-6.86697	7.64649	6.72106	3.42192e+06
4.13266	-1.32993	-6.76663	8.88722	5.40008	3.81759e+06
4.11738	-2.68317	-6.62968	10.0440	4.52731	4.16936e+06
4.10855	-4.10748	-6.45720	11.1068	3.92054	4.48039e+06
4.10246	-5.59908	-6.25089	12.0667	3.48356	4.75311e+06
4.09577	-7.15612	-6.01127	12.9074	3.16405	4.98732e+06
4.08570	-8.76745	-5.74349	13.6282	2.92564	5.18656e+06
4.07017	-10.4250	-5.45117	14.2237	2.74775	5.35181e+06
4.04750	-12.1200	-5.13825	14.6892	2.61681	5.48407e+06
4.01626	-13.8437	-4.80882	15.0191	2.52457	5.58336e+06
3.97522	-15.5861	-4.46750	15.2079	2.47135	5.64315e+06
3.92450	-17.3365	-4.11977	15.2623	2.44395	5.67471e+06
3.86415	-19.0857	-3.77022	15.1845	2.43824	5.68134e+06
3.79451	-20.8250	-3.42328	14.9777	2.46439	5.65112e+06
3.73505	-22.1451	-3.16214	14.7326	2.49432	5.61711e+06
3.66638	-23.5272	-2.89243	14.3923	2.55420	5.55088e+06

3.57542	-25.2046	-2.57179	13.8711	2.65903	5.44036e+06
3.47833	-26.8481	-2.26702	13.2430	2.80522	5.29671e+06
3.37629	-28.4537	-1.98051	12.5154	3.00174	5.12038e+06
3.26971	-30.0155	-1.71547	11.6909	3.26786	4.90747e+06
3.16067	-31.5340	-1.47207	10.7830	3.62582	4.65893e+06
3.05060	-33.0083	-1.25081	9.79986	4.11458	4.37347e+06
2.94090	-34.4384	-1.05186	8.74872	4.80019	4.04911e+06
2.83292	-35.8238	-0.875258	7.63563	5.80406	3.68234e+06
2.72822	-37.1646	-0.720907	6.46584	7.37048	3.26769e+06
2.62936	-38.4644	-0.586496	5.24841	10.0246	2.80192e+06
2.53877	-39.7252	-0.470527	3.98873	15.1810	2.27688e+06
2.45946	-40.9492	-0.371190	2.69172	27.9249	1.67878e+06
2.39606	-42.1372	-0.286779	1.36063	83.1051	973140.
2.35526	-43.2919	-0.214157	5.12227e-07	83.1051	146961.
2.35526	-42.1372	-0.286779	-1.36063	83.1051	146961.

END LOOP